Bachelor thesis in the course of studies
Audiovisual Media

Implementation of runtime debugging tools for the enterprise application environments Java EE and Spring

submitted by
Patrick Kleindienst

at Stuttgart Media University
on 20.08.2015

in partial fulfillment of the requirements for the degree of Bachelor of Engineering

First examinor: Prof. Dr. Simon Wiest
Second examinor: Dipl.-Ing. (FH) Christoph Diefenthal

Statutory Declaration

„Hiermit versichere ich, Patrick Kleindienst, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel *Implementation of runtime debugging tools for the enterprise application environments Java EE and Spring* selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.
Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.“

Place and date	Signature

Acknowledgement

Foremost, I would like to express my sincere thankfulness to my first advisor Prof. Dr. Simon Wiest for his assistance and his great support over the last months. This thesis highly benefits from his knowledge and his passion for programming, which he conveyed me during the first semesters of my studies at Stuttgart Media University.
Moreover, I want to thank my second advisor and team leader Christoph Diefenthal (Dipl.-Ing. (FH)), who gave me the opportunity to join his team as an intern, helped me with the topic selection and showed me his trust when he agreed to supervise this thesis. Particularly, I appreciated his willingness to support me instantly whenever there were unanswered questions concerning technical or formal issues.
I also want to thank all my other colleagues at *Bertsch Innovation GmbH* for having offered me a convenient and loose atmosphere, which has been an important condition for working on this thesis successfully. Especially, I want to set apart my room mates Markus Hettich (B.Sc.) and Frank Milde (B.Sc.), for having brought up the patience as well as the effort to support me countless times and provide me with their precious advice.
Besides, I also want to thank my former fellow student Johannes Zink (B.Eng.), who immediately agreed to proofread this thesis and supplied me with valuable feedback.
Lastly, I want to want to express my gratitude to my parents, for always believing in me and backing me in every moment of my life.

Abstract

When searching for bugs in Java enterprise applications, an essential part of the effort consists in redeploying the source code and relaunching the server over and over. In order to improve this situation, this thesis suggests the implementation of a *runtime debugging tool*. The tool's purpose is to facilitate the enrichment of operating application code with logging statements, which are inteded to generate additional output concerning the webapp's current state. On behalf of this so-called instrumentation, the actual process of debugging could be supported and accelerated without having to interrupt the server's execution.
Due to the significance of Java EE as well as Spring for today's enterprise development, the implementation of a dedicated debugging tool for each platform shall be covered. Both solutions pursue the same goal, but differ in the approach and the programming paradigm forming their basis. This document introduces their implementation details and evaluates them against a specification that defines the general conditions and expectations in terms of the capabilities of a satisfying result.

Zusammenfassung

Bei der Fehlersuche in Java-Enterprise-Anwendungen besteht ein wesentlicher Teil der Arbeit darin, den Quellcode immer wieder neu zu deployen sowie den Server neu zu starten. Zur Verbesserung dieser Situation schlägt diese Thesis die Implementierung eines *Laufzeit-Debugging-Tools* vor. Zweck dieses Tools ist es, die Integration von Logging-Statements in sich in Ausführung befindlichen Applikationscode zu ermöglichen. Diese Logging-Statements sollen zusätzliche Ausgaben erzeugen, die über den aktuellen Zustand einer Webanwendung Auskunft geben. Mithilfe dieser sogenannten Instrumentierung kann der eigentliche Debugging-Prozess unterstützt und beschleunigt werden, ohne dass die Ausführung des Servers unterbrochen werden muss.
Aufgrund der Bedeutung von sowohl Java EE als auch Spring für die heutige Enterprise-Entwicklung soll die Implementierung eines dedizierten Debugging-Tools für jede der beiden Plattformen behandelt werden. Beide Lösungen verfolgen dabei dasselbe Ziel, unterscheiden sich jedoch anhand von zugrunde liegendem Ansatz und Programmierparadigma. Dieses Dokument stellt die Details der Implementierungen vor und evaluiert diese gegen eine Spezifikation, welche die allgemeinen Rahmenbedingungen sowie die Erwartungen bezüglich der Einsatzmöglichkeiten eines zufriedenstellenden Ergebnisses definiert.

Contents

Abbreviations

AOP	Aspect-Oriented Programming
API	Application Programming Interface
CSS	Cascading Style Sheets
CTW	Compile-time Weaving
DI	Dependency Injection
DSL	Domain Specific Language
EJB	Enterprise JavaBeans
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
JAR	Java Archive
Java EE	Java Platform, Enterprise Edition
Java SE	Java Platform, Standard Edition
Javassist	Java Programming Assistant
JBoss AS	JBoss Application Server
JCP	Java Community Process
JDK	Java Development Kit
JMX	Java Management Extensions
JPA	Java Persistence API
JSF	JavaServer Faces
JTA	Java Transaction API
JVM	Java Virtual Machine

LDAP	Lightweight Directory Access Protocol
LTW	Load-time Weaving
MBean	Managed Bean
OOP	Object-Oriented Programming
PCW	Post-compile Weaving
POJO	Plain Old Java Object
POM	Project Object Model
RMI	Remote Method Invocation
RTW	Runtime Weaving
Spring AOP	Spring Aspect-Oriented Programming
SSH	Secure Shell
SSL	Secure Sockets Layer
VCS	Version Control System
WAR	Web Application Archive
WildFly AS	WildFly Application Server
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation

Troubleshooting and bugfixing belong to the main activities a developer has to face in his everyday life. A tool that is employed very often during the search of a flaw is the debugger, which allows to step through source code line by line (Oracle Corporation, 2014b).
However, using it when working on Java-based web applications is far from being comfortable. Due to Java's characteristics of a compiled programming language (Ullenboom, 2011) as well as the required server infrastructure for webapp development, the following steps have to be performed each time the modified code is prepared for another debugging cycle:

- *Compiling*: The updated source files have to be compiled into executable bytecode (Ullenboom, 2011).
- *Packaging*: The compiled code as well as other application components have to be packed as a ZIP file (e.g. in *Web Application Archive (WAR)* format) (Oracle Corporation, 2015n).
- *Deployment*: The generated artifact has to be handed over to the server, which takes care of the webapp to be installed (Apache Software Foundation, 2015d).
- *Restart*: Finally the application server has to be launched in debug mode.

Although modern *Integrated Development Environments (IDEs)* simplify this process by integrating widely used build tools like *Apache Maven* (Apache Software Foundation, 2015f) and supplying developers with several incorporated features (JetBrains s.r.o., 2015) for build and deployment management, their capabilities cannot deceive about the fact that performing these steps over and over is a very time-consuming and exhausting task.
Moreover, searching for the reason of an error on behalf of a debugger becomes even more complicated if the misbehavior consists for example in a performance issue, e.g. caused by inefficient and long-lasting database queries. Such problems often cannot be detected with a debugger and charge knowledge of how to handle

monitoring tools like *VisualVM* (Oracle Corporation, 2015r).
Regarding this issue from business management's point of view, every operating application that shows critical errors and needs to be shut down for troubleshooting is high in price for the company.
Therefore, reducing the time and effort needed to get a defective application back to work by speeding up the debugging and bugfixing process is a shared interest between the different divisions involved in a project.

1.2 Approach and goals of this thesis

The solution suggested by this thesis in order to accomplish faster debugging cycles and saving the need for stopping and restarting application servers over and over is the instrumentation of operating applications at their runtime. In theroy, this means that existing application code shall be enriched with logging statements, which are intended to produce additional output concerning information like the duration of database queries, parameter values belonging to certain method invocations or the current state of the members of a particular class. The generated output shall be gathered and stored in seperate log files for fast and convenient monitoring. The document calls this *runtime debugging*, since providing this kind of instrumentation by simultaneously preventing an application from going through the preparation steps described above constitutes the major advantage of this approach.
The research question this thesis deals with is related to the implementation of a tool that allows this kind of runtime debugging in the context of Java-based web applications. However, instead of focusing on merely one approach, it rather presents two different solutions, each being based upon another programming paradigm and tending to another enterprise application environment, i.e. *Spring* or *Java Platform, Enterprise Edition (Java EE)*. This facilitates a comparison of the possibilities and constraints when developing solutions for different platforms. These are the debugging tools which represent the results of the document's research:

- *Watson*: The first approach is a tool which is heavily based on the *Aspect-Oriented Programming (AOP)* paradigm (Lahres and Raman, 2015). It takes advantage of *Spring Aspect-Oriented Programming (Spring AOP)* (Pivotal Software Inc., 2015d), using the framework's capabilities to wrap particular method invocations with pre-defined debugging logic. According to its dependencies, its domain is restricted to Spring environments (Pivotal Software Inc., 2015g).

- *SherLog*: As an alternative solution, *SherLog* is built upon *Application Programming Interfaces (APIs)* which tackle the problem from another perspective. Instead of relying on AOP, it operates on an application's bytecode, integrating logging by directly modifying the class definitions loaded by a *Java Virtual Machine (JVM)* (Oracle Corporation, 2015p). Due to the tool's autonomy of any platform-specific libraries, it can be installed on every application server.

It is important to note that both *Watson* and *SherLog* do not pursue the goal of representing a medium which is able to completely redundantize manual debugging. The thesis rather offers proposals of how to accelerate and support this process in an effective manner, by introducing implementations that may help developers to cut down possible sources of error.

1.2.1 Fundamental requirements

In order to achieve a result that is eligible for being used in production environments, a final debugging tool has to meet several requirements. The following list depicts a specification that served as a starting point for the actual implementation:

- *Installation*: Integrating a debugging tool into an existing webapp's environment should be as simple as possible. Having to stop and restart a server to put the tool into operation is acceptable, but any additional configuration e.g. in *Extensible Markup Language (XML)* shall be avoided as far as possible.

- *Handling*: Just as the setup, working with such a tool ought to be considered preferably easy. The result should offer a *Graphical User Interface (GUI)* which allows convenient and intuitive handling, without presupposing deeper knowledge about the underlying project structure, i.e. package hierarchies and naming conventions. In this way, every employee could use it instantly without having to familiarize with the project or the debugging tool itself too long.

- *Granularity*: Java source code offers several locations in classes that qualify for being enriched with logging, like constructors or methods. Because method invocation stacks might be most important when investigating the causality of bugs, a final tool is expected to be able to instrument methods. Therefore, the thesis will not deal with the instrumentation of constructors or attributes.

- *Remoteness*: The debugging tool should be accessible both via a local as well as a remote host, without having to establish a *Secure Shell (SSH)* connection or using tools like *Microsoft Windows'* remote desktop application (Microsoft Corporation, 2015). This requires the choice of an adequate communication protocol.

- *Security*: Since the integration of logging statements implies an intervention into a running JVM, the tool must ensure that the application code cannot deliberately or unintentionally be altered in a malicious way. Ideally, a tool defines an abstraction layer that prevents a user from getting immediately in touch with the appliaction code. Furthermore, a secure channel for remote debugging must be provided.

- *Extensibility*: In order to provide a high level of flexibility, the result should offer dedicated interfaces which enable the seamless integration of additional customizations. A user must be able to extend the tool's capabilities according to his needs.

- *Platforms*: The debugging tool should be platform-agnostic as far as the Java enterprise environment is concerned. That means it is supposed to operate in standard Java EE ecosystems as well as in the context of alternative environments like Spring.

1.2.2 Evaluation of existing solutions

The fact that no existing alternative depicted a completely satisfying foundation brought about the desicion to put time and effort into the development of custom solutions. This part of the introduction provides a short overview of the tools and frameworks that have been examined at first and goes into their missing qualities.

JRebel

JRebel is a Java agent which can be installed in an application server. It is able to detect changes concerning webapp classes or resources and reloads them instantly without having to recompile and redeploy the whole application (ZeroTurnaround, 2015).
At first sight, JRebel already appears to be an appropriate solution for instrumenting a webapp with logging statements. However, using JRebel requires a developer to directly work on the application code in an IDE (ZeroTurnaround, 2015). But since the solution this thesis strives for should avoid direct code modifications, JRebel misses an essential criterion.
Besides, a developer or project leader that wants to apply JRebel on an application needs to have a corresponding working copy on his machine. Checking out a working copy from the *Version Control System (VCS)* and setting up a local workspace from scratch usually means additional effort and sometimes turns out to be more complicated as intended, because of barriers like missing access rights to repositories.
Above all, JRebel is not for free. In fact, *ZeroTurnaround* as the developing company offers 30 day trials(ZeroTurnaround, 2015), but being restricted to licenses is not a comfortable situation when searching for a tool to be used by the whole staff.

Byteman

Byteman is a bytecode manipulation tool developed and maintained by Red Hat Inc. (2015a). It allows the modification of Java code either at load-time or runtime, also redundantizing the need for recompiling and redeploying a webapp's code (Red Hat Inc., 2015a).
Looking at its features, Byteman also answers the purpose of integrating logging statements into existing code. However, using it requires to burrow into the tool's *Event Condition Action rules*. This is some kind of *Domain Specific Language (DSL)*, defining the additional logic that should be executed inside special *.btm* files (Red Hat Inc., 2015a). Getting familiar with Byteman's DSL is certainly possible, but prevents a client from being able to use it in no time.
Moreover, Byteman's appliance also requires knowledge about the application and its project structure. This breaches the requirement on an intuitive handling.

AspectJ

AspectJ is administrated by the Eclipse Foundation (2015) and provides a sophisticated AOP implementation. It constitutes an extension to the Java programming language and allows the enrichment of code either at compile- or load-time (Eclipse Foundation, 2003).
Although the AOP paradigm is of great importance as far as the *Watson* debugging tool is concerned, the AspectJ project does not feature any *Runtime Weaving (RTW)* capabilities (Eclipse Foundation, 2003). This results in a lack of flexibility at an application's runtime, which excludes it as a proper alternative.
Moreover, AspectJ's extension syntax poses a significant obstacle that prohibits the framework's instant usage. Even though the documentation assures that it is easy to learn, it takes time to incorporate its features (Eclipse Foundation, 2015).
In section 2.3.3, when introducing common AOP implementations, the thesis will revisit AspectJ and discuss some of its details.

1.3 Document structure

Before dealing with the implemented debugging tools in detail, chapter 2 gives an overview of the relevant frameworks and technologies which represent the foundations for both projects.
Chapter 3 then focuses on the *Watson* debugging tool, describing how to take advantage of Spring's internals in order to configure proxy objects dynamically.
Subsequently, *SherLog* is discussed in chapter 4. This part introduces an alternative approach and explains how to realize instrumentation on behalf of modifications on the bytecode level.
Chapter 5 picks up the requirements described above and examines if *Watson* and *SherLog* come up to these.
At least, chapter 6 provides an outlook on further improvements and possible extensions related to the status quo of both projects.

Chapter 2

Technologies and Foundations

This part of the thesis introduces the different technologies and frameworks which are essential for the discussion of how both debugging tools *SherLog* and *Watson* are implemented. It covers the very basics as far as they are relevant for being able to follow the details of the approaches presented in the subsequent chapters. Because examining these topics in detail is beyond the scope of this document, special issues related to these subjects will be deepend later on if necessary.

2.1 Java Platform, Enterprise Edition

Java EE provides a specification for a full-stack environment concerning the development of Java-based enterprise applications. It combines several technologies and APIs like *Java Persistence API (JPA)*, *Enterprise JavaBeans (EJB)* and *JavaServer Faces (JSF)* (Oracle Corporation, 2015f). The evolution of Java EE is liable to the *Java Community Process (JCP)*, which is made up of "industry experts, commercial and open source organizations, Java User Groups, and countless individuals", according to Oracle Corporation (2015g). Deeper knowledge concerning Java EE is not required, but it is a crucial issue to appreciate the coexistence of both the official specification and suitable implementations, as well as alternative projects like the *Spring Framework*.

2.2 Spring Framework

The Spring Framework was originally created to make Java enterprise development easier, addressing the drawbacks of earlier Java EE releases, e.g. the former complexitiy of EJB. In the meantime, Spring has reached it's fourth major release and has enveloped into a rich and popular ecosystem, offering lots of projects for Java-based web development and going beyond standard features like *Dependency Injection (DI)* (Walls, 2014, pp. 3-4).
A complete reference on current Spring features and projects can be found at the official homepage `https://spring.io/projects` (Pivotal Software Inc., 2015e).

2.2.1 Setting up an application context with Spring 4.0

Whether a Spring application is designed as a standalone application or executed inside a web container like *Tomcat* (Apache Software Foundation, 2015c), it needs an environment to manage the beans the application is made up of. Since Spring 4, such an application context can be defined using plain Java instead of being restricted to XML (Walls, 2014, p. 9). Listing 2.1 shows an example:

```
1 @Configuration
2 public class SpringSampleConfig {
3
4 @Bean
5 public MyBean bean() {
6   return new MyBean();
7 }
8 }
```

Listing 2.1: Java-based application context definition since Spring 4 (derived from Walls (2014, p. 9))

Note that a Spring configuration class has to be annotated with *@Configuration*. Its inherent beans are defined through producer methods marked with the *@Bean* annotation. Each Spring producer method instatiates a certain bean through an appropriate constructor. The invocation is conducted as soon as the Spring context is launched, whereby the defined beans are created and added to the Spring container. This container is a special enviroment where all beans are preserved by the framework. After a bean is completely initialized, it becomes available to the other beans existing inside the application container (Walls, 2014, p. 9). Throughout the next chapters, this thesis entirely relies on Java-based context configuration and therefore resigns XML.

2.2.2 Relationship of Spring and Java EE

Emphasizing the differences between Spring and Java EE, it is still important to note that Spring is not completely independent from Oracle's enterprise standards. As Gierke (2015) indicates, it also partially builds upon Java EE specifications and offers implementations for some of its APIs, for example *Servlet 3.1*, *Java Transaction API (JTA) 1.2* or *JPA 2.1*.

2.3 Aspect-Oriented Programming

The AOP paradigm is an essential topic regarding the *Watson* debugging tool. The following section highlights the fundamentals and points out the basics that *Watson's* implementation is based upon.

2.3.1 Motivation

As Walls (2014, pp. 97-98) notices, every software project depends on features that are needed in multiple parts of the application, like authentication, transactions or logging. These features are what the AOP paradigm calls *cross-cutting concerns*.
Following an unsophisticated approach, the implementation of such concerns would be repeated again and again at different spots anywhere in the application, although it always performs the same steps. Bulding an application this way is possible, but results in a hardly maintainable code base where changes become very elaborate, since desired adjustments have to be applied to multiple lines of code in diverse files. If the worst comes to the worst, this strategy leads to bugs breaking the entire application (Walls, 2014, pp. 97-98).
Of course one can argue that the *Object-Oriented Programming (OOP)* paradigm offers concepts like modularization and re-use of code snippets that would otherwise massively expand the application code. But nevertheless, these modular logging or transaction services must always be invoked explicitly whereever their functionality is needed. Even though profiting by the benefits of OOP, this code design would suffer from similar flaws, endangering the application's health if the encapsulated services are not used as intended by their creators. Moreover, the business logic gets polluted with non-functional code and maintainability can barely be guaranteed (Walls, 2014, pp. 97-98).
The AOP paradigm tackles this problem by not only isolating cross-cutting concerns into seperate modules, but also extracting their invocations from the business code they're affecting. Instead, this code is *woven* into the application either when an affected class is compiled, as soon as it is loaded, or even later at runtime. As a consequence, the developers are no longer responsible for explicitly implementing logging or security methods calls. The task to integrate these so-called *aspects* into business code is performed by application servers or appropriate frameworks (Walls, 2014, pp. 97-99; Lahres and Raman, 2015).

2.3.2 Aspect components and terminology

The following paragraphs describe the basic concepts of AOP. All definitions refer to the publication of Lahres and Raman (2015).

Interceptors

An *interceptor* is a single method that encapsulates an arbitrary cross-cutting concern's logic. Interceptors are categorized in *before-, after-* and *around* methods, indicating the point in time when they're invoked relating to the business code wrapped by an aspect.

Join points

Join points define types of spots in the code where an interceptor invocation may be integrated. They refer to method executions in the first place, but also cover e.g. constructors and attribute access. It depends on the framework or container in use whether a certain join point type is actually supported.

Pointcut

A *pointcut* defines a set of one or more join points that are supposed to trigger the execution of interceptors. In other words, pointcuts are a subset of the available join points, choosing them to plug in additional interceptor logic. Join points and pointcuts are in a many-to-many relationhip, meaning that not only a pointcut may encase more than one join point, but a single join point may match multiple pointcuts, too.

Advice

Being able to encapsulate additional logic in interceptors and choose pointcuts as interception candidates, some construct is needed to determine when exactly to call the interceptor method as soon as a matching pointcut is passed. That is what an *advice's* function is. It includes an interceptor's functionality and checks if it has to be invoked either before, after or around a specified pointcut.

Aspect

An aspect is a special type of class where both pointcuts and advice are defined and attached to each other. Figure 2.1 outlines how the interaction between the introduced AOP components works.

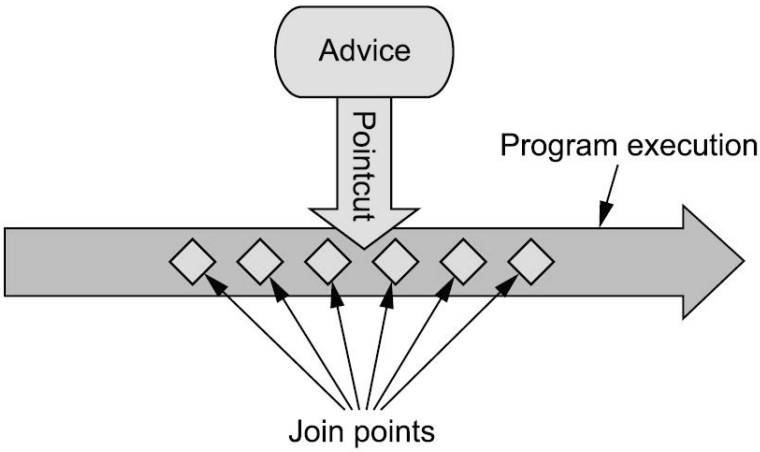

Figure 2.1: Illustration of how aspect functionality is woven into application code (taken from Walls (2014, p. 99))

2.3.3 AOP implementations

Adding AOP capabilities to a Java application can be achieved in more than one way. This thesis narrows down the scope of existing implementations to the most common solutions.

AspectJ

AspectJ is developed and provided by the Eclipse Foundation (2015). It represents a sophisticated AOP framework which supports several weaving techniques (Eclipse Foundation, 2015).
One such mechanism, which is called *Compile-time Weaving (CTW)*, requires a special compiler named *ajc*. This compiler is able to produce class files that already contain the additional aspect code (Eclipse Foundation, 2003).
Furthermore, AspectJ offers *Post-compile Weaving (PCW)* as an alternative approach. It is employed in case that aspects have to be woven into classes that are already existent in a binary form (Eclipse Foundation, 2003).
Alongside CTW and PCW, AspectJ also maintains *Load-time Weaving (LTW)*, which takes place as soon as a certain class is loaded by the JVM (Eclipse Foundation, 2015).
Another great advantage when using AspectJ is it's wide range of available join points, facilitating the definition of fine-grained pointcuts (Walls, 2014, p. 103).
However, a major disadvantage of AspectJ is its missing support for *RTW*, specifying the weaving of classes already loaded and defined by the JVM. The official documentation explicitly underlines that the current AspectJ version does not comprise this feature (Eclipse Foundation, 2003).

Spring Aspect-Oriented Programming

The Spring AOP project belongs to Spring's core modules (Walls, 2014, pp. 97-128; Pivotal Software Inc., 2015d). It has been developed upon the well-known proxy pattern, as described by Gamma, Helm, and Johnson (2001, pp. 254-267). These proxies take care of integrating interception code, forwarding the execution of the actual business logic to the original Spring beans. Since Spring AOP proxies are not instantated until a Spring application is launched, there is no need for an extra compiler that modifies class files already at compile time. Instead, it uses RTW as its only strategy, creating a proxy as soon as an affected bean is requested from the context for the first time (Walls, 2014, p. 103).
An obvious drawback might be the absence of available join points, since Spring AOP is restricted to method interception (Walls, 2014, pp. 102-103).
As for the implementation of the *Watson* debugging tool, this thesis sticks to Spring AOP and does not deal with AspectJ any further. On the one hand, AspectJ's missing support for RTW means a significant lack of flexibility, preventing indispensable modifications of running applications. On the other hand, Spring's AOP support and its focus on method interception turned out to be sufficient for enabling *Watson* to break into method definitions and insert logging statements.

2.4 Spring Aspect-Oriented Programming

As seen above, the Spring AOP project poses a lightweight alternative to AspectJ. It picks up the basic concepts of AOP, relieving its users from concentrating on all the fine-grained details of this programming paradigm and offering a more declarative approach based on annotations or XML. At the time the thesis was written, Spring AOP has been available in its second major version (Pivotal Software Inc., 2015d).

2.4.1 Demarcation from AspectJ

As Walls (2014, pp. 97-128) points out, some features of Spring AOP are inspired by AspectJ. For example, the way how aspects are defined is very similar, making use of AspectJ-specific annotations in order to declare a class as an aspect (Walls, 2014, pp. 97-128).
In contrast to AspectJ, Spring AOP follows a purely proxy-based approach for putting AOP into practice, although that limits its join point capabilities to method interception. But otherwise, realizing AOP features this way keeps a framework from depending on special compilers, that have to take care of bytecode supplement (Walls, 2014, pp. 97-128).
Moreover, since Spring AOP relies on Plain Old Java Objects (POJOs) for aspect definitions, there is no need for special language extensions or particular IDE plugins as they exist for AspectJ (Walls, 2014, pp. 106-107). Thus, using Spring AOP causes minimal overhead, assuming that method interception satisfies a user's needs.

2.4.2 Appliance of the proxy pattern in Spring AOP

In order to complement a bean's methods with additional capabilities, Spring AOP wraps it with a surrogate object (Figure 2.2). Such an AOP proxy intercepts calls to the inherent target bean, performing its tasks and subsequently handing off the invocation to the original bean. Of course the order of these incidents depends on which type of advice is about to be applied (Walls, 2014, p. 103).
Spring AOP allows the usage of *JDK dynamic proxies* (Johnson and Hoeller, 2015e) as well as *CGLIB proxies* (Johnson et al., 2015). Whereas default JDK proxies are capable of proxying one or more Java interfaces, CGLIB proxies are limited to concrete classes and hence are based on subclassing the target bean (Pivotal Software Inc., 2015d).
The proxy setup occurs once a bean that needs to be intercepted is instantiated and added to Spring's application context. That means that all the other beans that depend on the proxied bean never acquire a reference to the target bean itself, but are injected a proxy reference by the framework. As already mentioned above, this mechanism is categorized as RTW, since it happens after a Spring application has been launched (Pivotal Software Inc., 2015d).

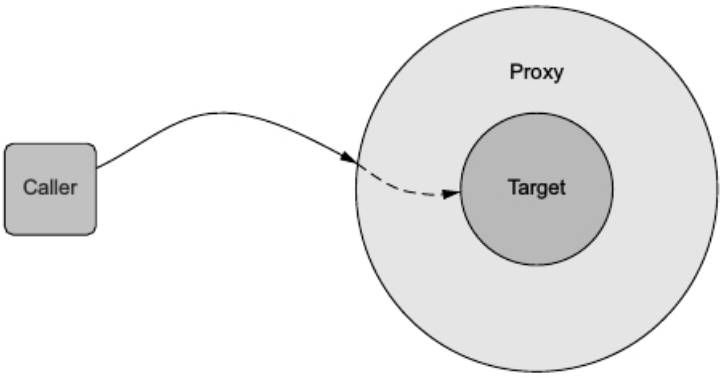

Figure 2.2: Usage of the proxy pattern in Spring AOP, as seen in *Spring in Action* by Walls (2014, p. 103)

2.4.3 How Spring produces AOP proxies

For managing the creation of proxy objects, the Spring Framework makes use of one of the extension points a Spring application container offers for enabling a client to interfere the creation of beans as well as the application context itself. The extension point used by Spring AOP is defined through the *BeanPostProcessor* interface, which specifies two callback methods, as shown in Listing 2.2.

```
1  public interface BeanPostProcessor {
2   Object postProcessBeforeInitialization(Object bean, String
       beanName) throws BeansException;
3
4   Object postProcessAfterInitialization(Object bean, String
       beanName) throws BeansException;
5  }
```

Listing 2.2: Spring's BeanPostProcessor interface (authored by Hoeller (2015b))

These callback methods are invoked by the framework in correlation with the lifecylce a Spring bean goes through (Figure 2.3).

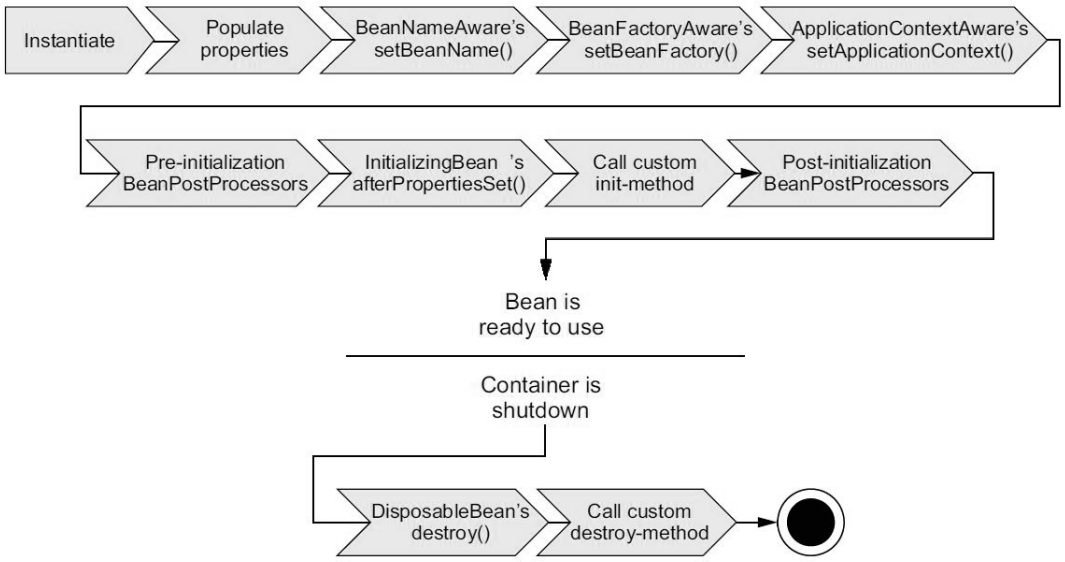

Figure 2.3: Lifecycle of a Spring bean, as seen in *Spring in Action* by Walls (2014, p. 20)

The diagram above shows every callback method that may or may not be called on a bean which is currently in creation. Besides the ones defined by different BeanPostProcessors, a Spring bean could also make use of further callbacks if the relevant class implements at least one of Spring's *Aware* interfaces (Beams, 2015). Amongst other things, their purpose is to make the container pass references on special objects to a bean through the methods they declare (Beams, 2015).
An essential point depicted by the figure is that the BeanPostProcessor callbacks from Listing 2.2 are applied directly before a newly created bean is initialized and accordingly after the initialization is finished. Thus, a bean could be hidden behind a proxy right before it is added to the application context and becomes available for DI (Pivotal Software Inc., 2015b).
Depending on how Spring aspects are defined, the framework provides several concrete BeanPostProcessor implementations for proxification, e.g. *AnnotationAwareAspectJAutoProxyCreator* in case of aspects declared on behalf of AspectJ's

annotation style (Johnson and Hoeller, 2015c). This class will be introduced during the next chapter.
It is also possible to use custom BeanPostProcessors in order intercept the bean fabrication process. On application context startup, Spring automatically scans the classpath for existing implementations of the interface and registers them for employment (Pivotal Software Inc., 2015b).

2.4.4 Definition of custom aspects

Since Spring AOP 2.0, aspects and their components can be declared in plain Java, redundantizing the use of XML configuration files (Walls, 2014, p. 102). This thesis therefore completely resigns XML configuration of aspects.
Listing 2.3 shows the simple definition of an aspect with Spring AOP, using Java POJOs:

```
1 @Aspect
2 public class MyAspect {
3   // a sample Spring AOP aspect
4 }
```

Listing 2.3: Definition of an aspect with Spring AOP (derived from Walls (2014, p. 106))

The only additional piece of information required is the *@Aspect* annoation, which resides in the *org.aspectj.lang.annotation* package, indicating its AspectJ origin. According to Walls (2014, p. 100), this simply marks a POJO as a regular bean that should be considered an aspect definition. As for the setup of aspects, there is no AspectJ involved as the annotation's fully qualified name might suggest (Walls, 2014, p. 100).

2.4.5 Filtering join points with pointcuts

In addition to the aspect class itself, one has to define one or more code snippets that describe the functionality that should be woven into the business code and tell Spring AOP about suitable join points on behalf of a pointcut definiton (Walls, 2014, pp. 103-105).
Starting with the pointcut declaration, Spring allows to use empty instance methods as markers, as shown in Listing 2.4:

```
1 @Pointcut("execution(public * *(..))")
2 public void interceptPublicMethods {
3   // nothing to do here
4 }
```

Listing 2.4: Declaring a pointcut with AspectJ's expression language (derived from Walls (2014, p. 105))

This Listing defines an emtpy method annotated with *@Pointcut* from the *org.aspectj.lang.annotation* package, returning void and taking no arguments. This annotation accepts an expression as a parameter, specifying which join points should be affected by the pointcut. The dialect used here is also known as AspectJ's *pointcut expression language*. Walls (2014, p. 104) gives a quick overview about the most relevant elements of this special language.
In short, the expression in listing 2.4 means that every class containing at least one public method shall be affected by the pointcut. The name, return-value and the arguments it takes do not matter, which is embodied by the asterisk (Walls, 2014, p. 104).
For a more detailed view on AspectJ's pointcut expression language, consider the official Spring AOP documentation (Pivotal Software Inc., 2015d).
It should be mentioned that the usage of an extra pointcut method is not mandatory. One could also have passed the expression directly to a certain advice. The advantage of doing it this way is that the pointcut becomes reusable and does not have to be re-declared several times in the same aspect class (Walls, 2014, p. 108).

2.4.6 Encapsulation of interception code with advice

What is still missing at this point is the code whose execution should be triggered whenever a matching join point is passed by the current thread. An examplary code fragment is shown in Listing 2.5.

```
1 @Before("interceptPublicMethods()")
2 public void doSomethingBefore {
3   System.out.println("Before method execution..");
4 }
```

Listing 2.5: Declaring an advice with Spring AOP (derived from Walls (2014, p. 107))

The defined interceptor method is annotated with *@Before*, another annotation residing in the *org.aspectj.lang.annotation* package. From the same package, *@After* or *@Around* could alternatively have been chosen, depending on the desired execution time (Walls, 2014, p. 107).
The selected annotation gets passed the pointcut declared in Listing 2.4, determining the method to be a before advice in this case. That means that the code contained in the interceptor method above is executed right before the application code inside a method matching the annotation's pointcut argument (Walls, 2014, pp. 108-109).

2.4.7 Bringing Spring AOP into action

Spring AOP and its proxy mechanism are not activated by default. Enabling it requires a few lines of configuration, either provided in plain Java or XML (Walls, 2014, p. 109).
Because only the Java configuration will be relevant when focusing on *Watson* in the next chapter, this section limits itself to this setup variation.

```
1 @Configuration
2 @EnableAspectJAutoProxy
3 public class MyJavaConfig {
4 }
```

Listing 2.6: Enabling Spring AOP using plain Java (derived from Walls (2014, p. 109))

As Listing 2.6 underlines, adding the *@EnableAspectJAutoProxy* annotation to an exsisting Spring configuration class is adequate to make Spring AOP understand all AspectJ annotations and put its proxies into action. As for the configuration of the aspects, they are treated and defined as ordinary Spring beans by instantiating them inside the configuration class (Walls, 2014, p. 109).

2.5 Instrumentation API

In order not to be restricted to Spring environments, the *SherLog* debugging tool resorts to the *Instrumentation API*, which is part of the *Java Platform, Standard Edition (Java SE)* specification since version 1.5. Its *Instrumentation* interface provides numerous methods which allow clients to modify existing class definitions for monitoring and instrumentation purposes (Oracle Corporation, 2015e).
The official documentation particularly underlines that the Instrumentation API as a powerful medium follows a strict security policy, prohibiting for example the addition or removal of fields or methods (Oracle Corporation, 2015e).

2.5.1 Advantages over plain class reloading

Although updating a class definition could also be achieved by creating a new class loader and reloading a modified version of a class, this workaround is not sufficient for the implementation of the debugging tool. Because this approach results in the creation of a new class, it does not affect objects that have already been inferred from the original class (Liang and Bracha, 1998).
Though, since e.g. singleton beans are only instantiated once when a deployed webapp is launched, accomplishing the customization of existing objects is a base requirement. In contrast to simple class reloading, the Instrumentation API is able to perform updates on exiting class definitions as well as their pertaining objects (Oracle Corporation, 2015e).

2.5.2 The -javaagent option

Looking for concrete implementations of the Instrumentation interface, the *Java Development Kit (JDK)* provides a single class called *InstrumentationImpl* (Oracle Corporation, 2004). But since its constructor is declared private, a client can not simply create his own instances of that class. Instead, only the JVM itself has the permissions to instantiate and manage Instrumentation objects (Oracle Corporation, 2015h).
As Oracle Corporation (2015h) points out, an application has to be supplied with an *agent* in order to be able to receive an Instrumentation instance.
An agent is defined as a self-contained application, which is packaged up as a Java Archive (JAR) and passed to the JVM via command line, in case that the agent should be executed right before regular application startup (Oracle Corporation, 2015h):

```
1 -javaagent:jarpath[=options]
```

Listing 2.7: Starting a Java agent via JVM parameter, as explained by Oracle Corporation (2015h)

Notice that starting a Java agent does not necessarily have to take place at the beginning of the JVM's startup. It is also an option to run it after an application has been launched, which requires the agent to be attached to a JVM programmatically. Therefore, the JDK features classes and methods to implement agents that get invoked at a later date (Oracle Corporation, 2015h).
As for this thesis, the focus will be on command line agents, because *SherLog* is intended to be available right after the application server's initiating sequence is finished.

2.5.3 Setting up a Java agent

Regardless of which kind of agent startup is chosen, the components that have to be included in the agent's JAR are always the same (Oracle Corporation, 2015h):

- a manifest file named *MANIFEST.MF*, specifying the agent's configuration
- a class definition, containing a static *premain*/*agentmain* [1] method

[1] Method has to be named *premain* if the agent is launched via command line, *agentmain* otherwise

The MANIFEST.MF file

This file contains meta information defining several conditions for the agent's execution. The definition of the properties occurs as a set of key-value pairs seperated by a colon (Oracle Corporation, 2015h).
This section will not cover all the available properties and values, rather focusing on the ones which are essential for *SherLog*:

- *Manifest-Version*: The version of the manifest
- *Premain-Class*: Indicates the name of the class where the agent's execution starts. This class must declare the *premain* method.
- *Can-Retransform-Classes*: Defines whether class transformations ought to be allowed or forbidden and must be assigned a boolean value.

Defining a premain method

The agent's premain callback method is where the JVM's Instrumentation instance can be received. It comes with two possible signatures, comprising a *String* of agent arguments in any case, as well as an optional Instrumentation argument (Oracle Corporation, 2015h):

```
1 public static void premain(String agentArgs, Instrumentation
      instr)
2 public static void premain(String agentArgs)
```

Listing 2.8: Signatures of the premain method (Oracle Corporation, 2015h)

Note that this method is always invoked by the JVM itself, which is why the document refers to it as a callback method.

2.5.4 Working on class definitions with the Instrumentation interface

With an Instrumentation instance at hand, it is possible to induce modifications concering currently loaded classes. This object can be considered a handle that serves as a cutting point for triggering different actions and customizations on class definitions, while the JVM always assures that the application's security is not compromised (Oracle Corporation, 2015e).

Gathering JVM setup information

Besides facilitating modifications on the bytecode level, the Instrumentation API offers a set of methods for retrieving information about the JVM's current state. For example, the Instrumentation object can be asked for classes that are currently known to the JVM, as shown in listing 2.9 (Oracle Corporation, 2015e).

```
1 public Class[] getAllLoadedClasses()
```

Listing 2.9: Retrieving an overview of currently loaded classes (taken from the Instrumentation interface, Oracle Corporation (2015h))

Moreover, it can be used to check if the properties defined in MANIFEST.MF have been applied successfully:

```
1 public boolean isRetransformClassesSupported()
2 public boolean isRedefineClassesSupported()
```

Listing 2.10: Checking the JVM configuration (taken from the Instrumentation interface, Oracle Corporation (2015h))

The differences between retransforming and redefining a class are discussed in the following section.

Retransforming application classes

When the Instrumentation object's *retransformClasses* method is invoked, it iterates on a list of *ClassFileTransformer* objects and applies them one by one, invoking their *transform* callback (Oracle Corporation, 2015b).
This process always starts from the initial class files as they have been loaded on the application's startup. Thus, the procedure requires the original class definitions to be cached by the JVM (Oracle Corporation, 2015e).

```
1 public void retransformClasses(Class<?>... classes) throws
      Exception
```

Listing 2.11: Retransforming a set of classes (taken from the Instrumentation interface, Oracle Corporation (2015h))

Redefining application classes

Alternatively, it is possible to go beyond the retransformation classes and redefine them from scratch. In this way, the current definition of one or more classes can be replaced completely. As a result, the former class definitions are thoroughly dropped and deleted (Oracle Corporation, 2015e).

```
1 public void retransformClasses(Class<?>... classes) throws
     Exception
```

Listing 2.12: Redefinition of classes (taken from the Instrumentation interface, Oracle Corporation (2015h))

The redefinition of classes will not play a role in the context of the *SherLog* debugging tool. Since the JVM stores the original representations of its classes, retransforming them is sufficient for adding additional logging as well as resetting them to their initial state as soon as the instrumentation code is no longer needed (Oracle Corporation, 2015e).

Applying ClassFileTransformes

As mentioned above, the Instrumentation API is based on the concept of ClassFileTransformers for altering existing class definitions. A client has to provide a concrete implementation of this interface by determining what should be done when its transform callback is invoked. This method defines the following signature (Oracle Corporation, 2015b):

```
1 public byte[] transform(ClassLoader loader,
2                         String className,
3                         Class<?> classBeingRedefined,
4                         ProtectionDomain domain,
5                         byte[] classFileBuffer) throws
                             IllegalClassFormatException
```

Listing 2.13: Callback method as declared by the ClassFileTransformer interface (Oracle Corporation, 2015b)

The *byte* array parameter is where the existing class definition is passed to the ClassFileTransformer as a binary representation. This array can be modified using an arbitrary bytecode engineering framework like *Java Programming Assistant (Javassist)* (Chiba, 2015).
Afterwards, the client does not have to care about the adjustments being applied. Instead, the modified byte array simply has to be returned and the Instrumentation API handles the registration of the updated class definition (Oracle Corporation, 2015b).

Before a custom ClassFileTransformer can be applied in transformations, it has to be registered at the Instrumentation instance. For that purpose, an appropriate method is provided (Oracle Corporation, 2015e):

```
1 public boolean addTransformer(ClassFileTransformer
      transformer)
```

Listing 2.14: Registering a ClassFileTransformer at the Instrumentation object (Oracle Corporation, 2015b)

A ClassFileTransformer can also be removed if it shall not be adopted during a successive transformation:

```
1 public boolean removeTransformer(ClassFileTransformer
      transformer)
```

Listing 2.15: Removing a ClassFileTransformer from the Instrumentation object (Oracle Corporation, 2015b)

2.6 Java Programming Assistant

Javassist is a bytecode engineering library which has originally been created by Chiba (2015) and is maintained by *JBoss*. It enables developers to operate on Java bytecode at runtime, either in high- or low-level mode. The high-level API, also called source-level API, facilitates editing class files without deeper knowledge of how bytecode actually looks like. On the other hand, Javassist's low-level API offers a possibility to work on plain bytecode without the presence of any abstractions (Chiba, 2015).
The *SherLog* debugging tool takes advantage of Javassist by using it to insert logging statements into methods of existing class definitions. As already demonstrated, their binary class representations are accessible on behalf of the Instrumentation API (see section 2.5).
The subsequent explanations are based on the official Javassist tutorial (JBoss Inc., 2015a).

2.6.1 The Javassist CtClass

When compiling Java source code, the resulting bytecode is stored in class files. In order to access this bytecode, Javassist uses instances of one of its inherent classes, called *CtClass* (= compile-time class). Instances of this class can be considered a handle for accessing the content of a class file and working on it (JBoss Inc., 2015a).

2.6.2 The Javassist ClassPool

Javassist makes use of a *ClassPool* for managing its CtClass instances. A ClassPool has the properties of a hash table, recording CtClass objects and making them available through a unique key derived from the fully qualified class name of the referenced Java class. For retrieving a certain CtClass object, the ClassPool class offers a *get* method (Listing 2.16), taking the associated key as a parameter (JBoss Inc., 2015a).

```
1 ClassPool classPool = ClassPool.getDefault();
2 CtClass ctClass = classPool.get("test.MyClass");
```

Listing 2.16: Obtaining a CtClass instance from a ClassPool (derived from JBoss Inc. (2015a))

If no existing CtClass instance can be found, the ClassPool scans the class path for a class file matching the parameter's value and constructs a new CtClass object, given that a suitable file has been found. The newly created object is immediately recorded in the pool, enabling later access without having to reload the bytecode from the file again (JBoss Inc., 2015a).
Javassist uses the paths of the JVM's underlying system class loader as well as its child loaders in order to find class files. When running inside an application server, a ClassPool may not be able to locate some classes, since these servers use several class loaders because of isolation concerns. Therefore a ClassPool can be announced additional paths where it should look for class files, as shown in Listing 2.17 (JBoss Inc., 2015a).

```
1 ClassPool classPool = ClassPool.getDefault();
2 classPool.insertClassPath("C:\Users\testUser\appClasses");
```

Listing 2.17: Adding an additional class path to a ClassPool (derived from JBoss Inc. (2015a))

2.6.3 Enhancing method bodies

What *SherLog* needs Javassist to do is to integrate additional instrumentation code fragments at the beginning as well as at the end of a method. Benefiting from Javassist's high-level API, this takes just a few lines (Listing 2.18). The *CtMethod* class provides the necessary operations for this purpose (JBoss Inc., 2015a):

```
1 CtMethod ctMethod = ctClass.getDeclaredMethod("testMethod");
2 ctMethod.insertBefore("System.out.println(\"Before\");");
3 ctMethod.insertAfter("System.out.println(\"After\");");
```

Listing 2.18: Adding some code to a method's body (derived from JBoss Inc. (2015a))

At least, the bytecode modifications have to be commited, so that they can be applied in the application. Javassist offers several alternatives to achieve this. This thesis will only make use of returning the raw bytecode of a modified class (Listing 2.19), since updating existing class definitions is left to the Instrumentation API, as described in section 2.5.1.

```
1 byte[] bytes = ctClass.toBytecode();
```

Listing 2.19: Receiving the updated bytecode of a class on behalf of CtClass (JBoss Inc., 2015a)

2.7 Java Management Extensions

The Java Management Extensions (JMX) technology allows the monitoring of Java applications as well as their underlying JVMs. To make this possible, it uses special objects called *Managed Beans (MBeans)*, which are exposed by a *MBean server*. Every MBean is part of the webapp itself and defines operations that can be invoked remotely from outside of the running application. For this purpose, every JDK comes with a tool called *JConsole*. This tool provides a rudimental GUI for simple access to the resources published by an application as well as its JVM (Oracle Corporation, 2015j).
JMX employs several connectors for establishing connections to available MBean servers. These connectors rely on different communication protocols like *Remote Method Invocation (RMI)* (Oracle Corporation, 2015q).
Which protocol is actually used for interacting with an MBean server does not make a difference, since the JMX APIs stashes these settings behind uniform interfaces (Oracle Corporation, 2015j).

2.7.1 Implementing a MBean

The MBean interface

According to Oracle Corporation (2015j), MBeans are defined as ordinary Java classes backed by an interface which determines the attributes and operations that are intented to be published by the MBean server. The JMX specification requires this interface to bear the name of the class that implements it, followed by a *MBean* suffix.

Assuming that a MBean class is named *JmxService*, a suitable interface could look like this:

```
1  public interface JmxServiceMBean {
2    public void doNothing();
3    public int getNumber();
4    public void setNumber(int number);
5  }
```

Listing 2.20: An exemplary MBean interface (Oracle Corporation, 2015o)

The specification also follows particular naming conventions to distinguish operations from attributes. A method starting with *get* and returning not *void* is considered a *getter*, whereas a method beginning with *set* and returning *void* is treated as a *setter*. Other signatures are interpreted as operations (Oracle Corporation, 2015o).

The MBean class

The MBean class implements the respective MBean interface and provides implementations for its method declarations:

```
1  public class JmxService implements JmxServiceMBean {
2
3    private int number;
4
5    public void doNothing() {
6      // nothing to do here
7    }
8
9    public int getNumber() {
10     return this.number;
11   }
12
13   public void setNumber(int number);
14     this.number = number;
15   }
16 }
```

Listing 2.21: An exemplary MBean class (Oracle Corporation, 2015o)

Registering a MBean at the MBean server

In order to be able to access MBeans as well as their operations and attributes, the instrumented application needs to start a MBean server and register them. The subsequent main method illustrates the required steps (Oracle Corporation, 2015o):

```
1 public static void main(String[] args) throws Exception {
2   MBeanServer mbs = ManagementFactory.
3                     getPlatformMBeanServer();
4   ObjectName name = new ObjectName("com.test:type=MyService")
5   JmxService mbean = new JmxService();
6   mbs.registerMBean(mbean, name);
7 }
```

Listing 2.22: Exemplary MBean registration (Oracle Corporation, 2015o)

Invoking *getPlatformMBeanServer* returns a reference to the MBean server that is currently in use. If no MBean server has been created yet, a new one is instantiated and started automatically. When launching an application server, it typically takes care of providing a MBean server that is ready for use without further ado (Oracle Corporation, 2015o).

2.7.2 Configuring JMX in Spring environments

The Spring Framework not only supports the JMX specification's standard, but also provides several pre-defined beans which make it easy to enable JMX in the context of a Spring application (Figure 2.4). Putting it into operation requires the configuration of three Spring beans in the application context (Walls, 2014, pp. 523-525):

- Spring environments also require an MBean server to make MBeans accessible from outside of the running JVM.
- A Spring MBean exporter takes care of registering one or more Spring beans at the MBean server. Exporters allow every Spring bean to be accessible via JMX without being forced to implement any specific interface.
- Finally, it takes at least one single Spring bean that is intended to be published in order to operate on it.

At this point, the thesis skips how the setup of the respective beans could look like and picks it up in section 3.4, when talking about *Watson's* JMX configuration.

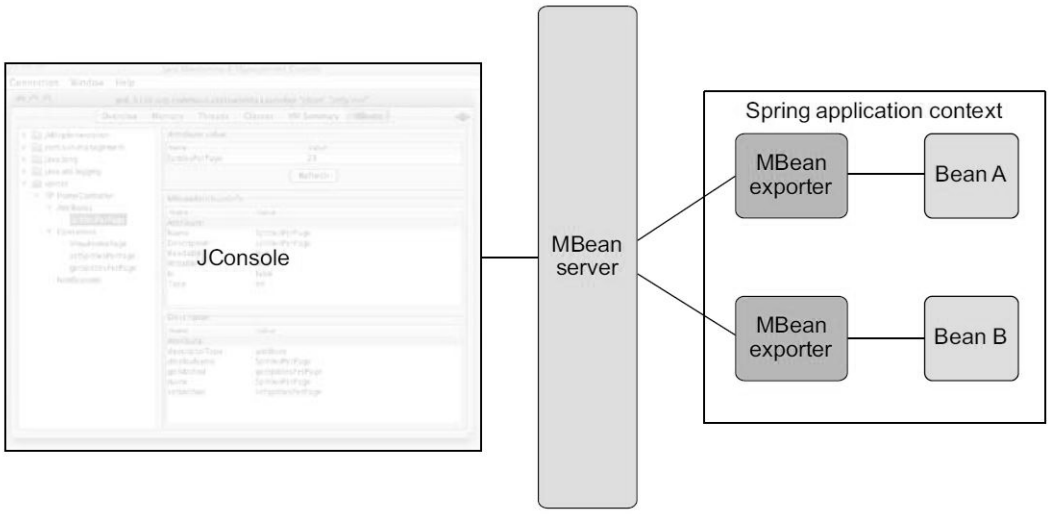

Figure 2.4: JMX in the context of a Spring application (taken from Walls (2014, p. 525))

2.7.3 JMX security

For the sake of completeness, the introduction ends the presentation of the JMX technology with taking a short look at its security mechanisms. Although IT security is of great importance for every company that relies on a server infrastructure, dealing with the supported protocols in detail is beyond the scope of the thesis. Nevertheless, it provides a brief dump of the features that facilitate the protection of applications which make use of JMX (Oracle Corporation, 2015l):

- *File-based password authentication*: JMX enables access control by means of file-based password authentication (Oracle Corporation, 2015l).
- *Lightweight Directory Access Protocol (LDAP) authentication*: As an alternative to the usage of files as password storages, JMX also supports the incorporation of a LDAP server for user authentication (Oracle Corporation, 2015l).
- *Encryption*: In order to secure communication data, JMX offers the possibility to enable encryption on behalf of the *Secure Sockets Layer (SSL)* protocol (Oracle Corporation, 2015l).

For more information on the security settings as well as on how to enable or disable them, consider Oracle Corporation (2015l).

2.8 Apache log4j

Apache log4j is a logging library provided by the Apache Software Foundation (2015a). It offers a lightweight configuration through properties files and supports logging to different sinks, such as files or java.io.Writers (Apache Software Foundation, 2015a). Both *SherLog* and *Watson* rely on *Apache log4j* when it comes to enhance application classes with additional logging capabilities.

Chapter 3

Instrumenting application code with *Watson* and Spring AOP

This chapter covers the *Watson* debugging tool and examines the details of how it takes advantage of Spring AOP to serve its purpose of complementing a running application's code with additional logging.
The presentation of essential technologies already introduced the basics of Spring AOP as well as its fundamental usage. It can be anticipated that the framework's standard mechanism is not enough to reach the desired level of dynamic behavior and flexibility. Therefore, this part of the document immerges deeper into the Spring AOP project in order to elaborate a solution that converges the specification, that has been put in place at the beginning, as far as possible.
Instead of merely stepping through the code that realizes *Watson's* functionality, this chapter roughly traces the evolution of the tool, outlining the different phases of its development. It starts with a initial and naive approach, based on the manipulation of a bean's lifecycle. However, it moves along to another possibilty, focusing more on framework internals, once this original idea does not lead the project towards a reasonable solution.
The entire *Watson* source code is available on *Github* under
`https://github.com/Patrick-Kleindienst/BIG-Watson`.

3.1 Limitations of standard Spring AOP

Section 2.4 already demonstrated the core functionality of Spring AOP and how it implements the AOP paradigm. It allows the definition of aspect classes containing interceptors, advice and pointcuts for complementing existing business code with additional capabilities. The framework takes care of weaving in the aspect logic by creating proxy objects, which wrap the actual Spring beans as soon as they are created (Walls, 2014, pp. 101-103).
Although Spring's RTW mechanism seems to be a promising starting point for a debugging tool, its convenient usage relying on annotated classes and methods for aspect definitions is very cumbersome. The reason is that all the components declared in a class annotated with @Aspect are some kind of frozen after the process of compiling (Walls, 2014, pp. 102-103).
Unfortunately, this lack of flexibility could not simply be compensated by adjusting existing aspect classes on the bytecode level. Of course an aspect could be modified and reloaded on behalf of a new class loader. But since aspects are processed and prepared during the startup phase of a Spring application context (Walls, 2014, p. 103), this would only have an effect if the context would be prompted to recreate and reinitialize all the beans as well as their associated surrogate objects.
Indeed, following this strategy might lead to a practical result. However, refreshing a Spring application context and potentially deleting any server-side state that may has been stored in arbitrary beans does not make a clean impression.
Considering the mentioned characterisitics of Spring AOP as well as the outlined workaround clearly exposes the drawbacks of the framework's standard behavior. Below, this thesis will focus on two approaches for enabling the usage of Spring AOP and RTW in a more dynamic fashion, bypassing the static behavior the framework proves if applied conventionally.

3.2 A naive approach

The previous section already raised the idea of inducing the recreation and reinitialization of existing AOP proxies after having modified the bytecode of an aspect class. Getting this to work takes the respective Spring bean to be instantiated, wrapped in an updated proxy object and attached to the application context from scratch. This section examines how the configuration of a Spring application context can be modified to obtain this fashion of bean management behavior without having to refresh the context completely.

3.2.1 Understanding Spring bean scopes

The *scope* of a bean determines its life span. A Spring bean's life cycle starts as soon as it is instaniated, initialized and added to the application context, and ends once it is removed from the context and approved for garbage collection. Spring provides several scopes out of the box, whereupon the differences between

the *singleton* and the *prototype* scope are important as far as the debugging tool's development is concerned (Walls, 2014, p. 81).

The singleton scope

What makes up a singleton bean is that the Spring container creates and manages only one instance for a certain bean definition. This single instance is shared by the whole application as long as it is running. That means that whenever a singleton bean is requested from the application context, e.g. by means of the bean definition's unique id, the caller receives exactly the same object (Pivotal Software Inc., 2015c).
Since the same applies for AOP proxies wrapping singleton beans, there is no possibility to recreate them while the application is operating. The only resort in this case is to find a way to modify existing proxies, adapting their interception behavior to the current instrumentation needs. The thesis will deal with this idea when introducing a second, more sophisticated approach.

The prototype scope

A Spring bean with prototype scope is newly created every time it is requested from the application context (Pivotal Software Inc., 2015c).
Because this simultaneously results in the instantiation of a new AOP proxy object, assumed that the current bean has to be enriched with aspect capabilities, this scope is a suitable starting point for the first approach covered by this section.

3.2.2 Modification of proxies on behalf of the prototype scope

The first attempt of a debugging tool makes use of the fact that a prototype bean is recreated every time the application context is asked for it. It assumes that modifying an aspect's interception and pointcut bytecode right before triggering the reinstaniation of a bean and its corresponding proxy object leads to an AOP proxy whose behavior is adjusted to the preceding aspect modifications.
However, this idea does not respect the fact that most enterprise applications are not only made up of prototype-scoped beans, but also rely on singleton beans, e.g. for storing server-state that should be shared accross the entire application (Walls, 2014, p. 81).
If the debugging tool should be able to intercept a bean regardless of its designated scope, it has to impose the prototype scope on every bean before it is added to the context.

Customizing a bean's scope

As already explained in the previous chapter, a Spring container that houses an application's beans offers extension points to override existing bean instances as well as their configuration metadata. Since changing the scope of a bean definition affects the configuration of beans, the latter has to be adapted. For that

purpose, Spring defines the *BeanFactoryPostProcessor* interface. According to Pivotal Software Inc. (2015b), this interface declares a single method which allows an implementing class to join in the creation process of *BeanDefinitions*. A BeanDefinition contains inevitable information about a bean's characteristics, like the class name or its scope (Hoeller and Harrop, 2015a).
Listing 3.1 shows a BeanFactoryPostProcessor that checks if a BeanDefinition's scope property is set to *singleton* and changes it to *prototype*, if necessary.

```
1  public class PrototypeScopeBeanFactoryPostProcessor
       implements BeanFactoryPostProcessor {
2
3    @Override
4    public void postProcessBeanFactory(
       ConfigurableListableBeanFactory factory) throws
       BeansException {
5      for (String defName : factory.getBeanDefinitionNames()) {
6        BeanDefinition def = factory.getBeanDefinition(defName);
7        if (def.getScope().equalsIgnoreCase("singleton")) {
8            def.setScope("prototype");
9        }
10     }
11   }
12 }
```

Listing 3.1: A BeanFactoryPostProcessor for changing bean scopes

Whereas a BeanPostProcessor is invoked around a bean's initialization process, the execution of BeanFactoryPostProcessors takes place much earlier. Since they affect the bean definitions rather than operating on the instances itself, their work has to be finished before the first bean is instantiated (Pivotal Software Inc., 2015b).

Dealing with Spring's proxification decision caching mechanism

Testing the *PrototypeScopeBeanFactoryPostProcessor* indicates that it works as expected for beans which are already proxied when they are created for the first time. Further attemps proved that with this elaboration, it is not possible to make Spring building a proxy around a bean that was not considered a proxy-candidate when it was initially requested from the Spring context. Based on the fact that every bean has now prototype scope, this behavior seems strange at first sight. However, stepping through the AOP proxy creation process reveals that Spring AOP uses a sophisticated appendage in order to enable a high-performance preparation of proxy objects. But in opposition to what this formulation might suggest, Spring AOP does not cache real proxy instances. Instead, the framework keeps track of its decision whether a bean has to be proxied or not when it is instantiated for the first time. Thus, Spring can perform a look-up as soon as a bean is requested again and saves the need for checking it against the existing aspects

and their pointcuts. This mechanism might increase Spring's performance, but prevents a debugging tool based on this approach from instrumenting beans that were not initially registered as proxy-candidates.
Listing 3.2 shows an excerpt of the *AbstractAutoProxyCreator*'s *postProcessBeforeInstantiation* method (Hoeller, Johnson, and Harrop, 2015), which is responsible for evaluating if a bean has to be proxied as well as remembering this decision:

```
1  @Override
2  public Object postProcessBeforeInstantiation(Class<?>
       beanClass, String beanName) throws BeansException {
3    Object cacheKey = getCacheKey(beanClass, beanName);
4    if (beanName == null || !this.targetSourcedBeans.contains(
       beanName)) {
5     if (this.advisedBeans.containsKey(cacheKey)) {
6       return null;
7     }
8     if (isInfrastructureClass(beanClass) || shouldSkip(
        beanClass, beanName)) {
9       this.advisedBeans.put(cacheKey, Boolean.FALSE);
10      return null;
11    }
12    // initiating bean proxification
13 }
```

Listing 3.2: Spring AOP uses a key-value store for remebering proxification decisions (taken from Spring's AbstractAutoProxyCreator class, authored by Hoeller, Johnson, and Harrop (2015))

Since the concrete implementation of this abstract BeanPostProcessor subclass used for handling annotation-configured aspects, *AnnotationAwareAspectJAutoProxyCreator*, is not declared final, nothing prevents a developer from subclassing it and override the default behavior of Listing 3.2 (Johnson and Hoeller, 2015c). Although this means an intervention in the framework's core functionality, it constitutes a functional way to bypass the decision caching described above.

Spring's advisor cache

The decision whether or not to front a bean by a proxy is not the only information that is cached for later access. Spring also avoids that aspect classes have to be searched and converted into so-called *advisors*, implementations of the *Advisor* interface, over and over again whenever a proxy has to be built.
An advisor is a Spring-specific equivalent of a single interception method residing in an aspect class, accompanied by a certain pointcut expression. In other words, an advisor defines nothing but a concrete single aspect. Spring AOP creates them when processing detected aspect classes and adds them to proxy objects in order to configure their behavior (Pivotal Software Inc., 2015a). The thesis will come back to this point when dealing with the alternative implementation approach.

The *BeanFactoryAspectJAdvisorBuilder* class (Hoeller, 2015a), which encapsulates the task of processing annotated aspect classes, employs a list of such advisors. This list is only initialized when the class' *buildAspectJAdvisors* method is invoked for the first time. Listing 3.3 gives an overview of the most important lines of code, since the whole method is quite complex in its entirety:

```
1  public List<Advisor> buildAspectJAdvisors() {
2   List<String> aspectNames = null;
3   synchronized (this) {
4    aspectNames = this.aspectBeanNames;
5    if (aspectNames == null) {
6     /* search for aspect classes,
7        create corresponding Advisors
8        and store them in a list */
9    }
10   }
11  }
```

Listing 3.3: Spring also caches advisors to increase its performance (taken from Spring's BeanFactoryAspectJAdvisorsBuilder class, authored by Hoeller (2015a))

How the search for aspects as well as their caching is exactly implemented is not of particular importance. The code snippet only emphasizes the significance of the if-statement which determines whether the advisor setup procedure has to be executed. If the list of aspect names already exists, Spring AOP skips this job and returns a list of stored advisors (Hoeller, 2015a).
Once more a developer is free to override this by subclassing the class and providing a custom implementation, although this entails the adaption of further framework components to plug in the customized code.

3.2.3 Facing the problems raised by this approach

At this point, this thesis stops with the explanations of this first approximation towards the *Watson* debugging tool. Pausing and observing the arrangements that have been suggested so far, it turns out that this approach impacts the application as well as Spring AOP itself in a critical manner. This document therefore skips further details covering the missing bytecode modification of aspect classes and draws the attention on the issues raised by the implementation that has been introduced so far.
Considering the sections that dealt with the caching mechanisms used by Spring to optimize proxy creation, it is evident that overriding the framework's default behavior is a non-trivial and error-prone task. Only the statements responsible for caching may be removed, whereas a lot of code that is still required has to be copied and pasted. Beyond the risk of errors, modifying Spring's behavior this way raises the impression of hacking the framework's internals, due to the fact that it offers no obvious interfaces for customization. This kind of implementation is far from meeting the requirements of a clean code base that respects the intended

extension points provided by an API like Spring AOP.
Moreover, changing the scope of beans in order to instrument them poses a great intervention into an application's configuration. If the underlying webapp depends on shared state recorded by singleton beans, as already mentioned in section 3.1, it could become completely useless in the last resort.
Hence, the thesis moves on to another approach, which proves that the ambition of runtime debugging can be realized without applying questionable changes to the core of Spring AOP or an existing webapp configuration.

3.3 Implementing *Watson* by building advisors manually

The first attempt to build a highly dynamic debugging tool based on Spring AOP failed. Instead of forcing the application context to recreate a proxy each time a corresponding aspect is changed, *Watson* should concentrate on adjusting existing proxy objects without having to reinstantiate them afresh.
The thesis already went into Spring's Advisor interface briefly when it explained how the framework instantiates advisors based on located aspect classes, uses them for configuring proxy objects and stores them for later access. This describes the mechanism Spring uses in the background when complementing business objects with e.g. transactional capabilities and applying further user-defined aspects (see section 3.2.2).
Figure 3.1 illustrates how advisors integrate into the proxy-based chain of method invocation. The AOP proxy takes care of invoking matching advisors either before, around or after the respective method call (Johnson, Hoeller, and Colyer, 2015).

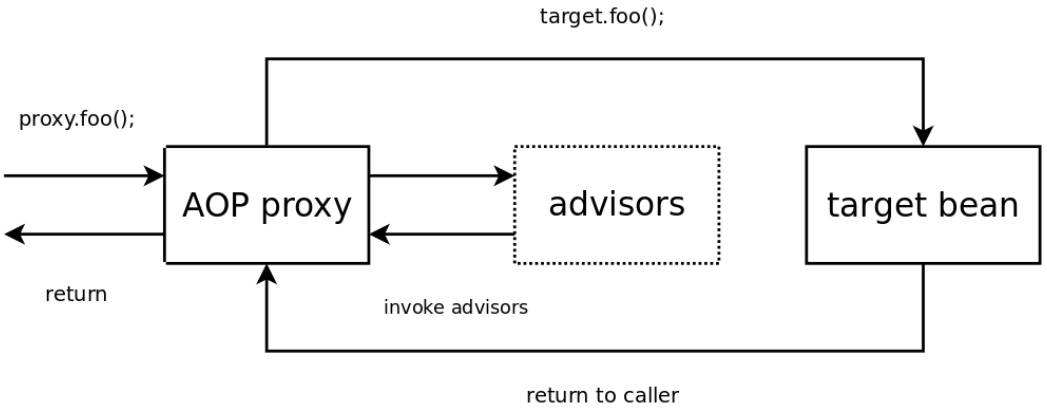

Figure 3.1: AOP proxies make use of advisors for intercepting method calls

What *Watson* needs at this point is some kind of interface that allows it to manipulate an AOP proxy's advisors. In the best case, this could be achieved by respecting Spring's internals and using only operations provided for this purpose.

3.3.1 The JdkDynamicAopProxy class

The search for configuration possibilities starts with the proxy classes itself. The introduction of technologies already mentioned that Spring AOP makes use of two different types of proxies, depending on whether their creation should be based on subclassing the target class or implementing its interfaces (Pivotal Software Inc., 2015d).
Since relying on JDK dynamic proxies is Spring's default behavior, the pertaining *JdkDynamicAopProxy* class serves as a starting point (Johnson and Hoeller, 2015e). The following concepts related to this class also apply to its CGLIB counterpart named *CglibAopProxy* (Johnson et al., 2015).

AOP proxy configuration

The first thing that attracts the attention is the parameter a JdkDynamicAopProxy gets passed via its constructor:

```
1 public JdkDynamicAopProxy(AdvisedSupport config) throws
     AopConfigException {
2  // setup proxy configuration
3 }
```

Listing 3.4: JdkDynamicAopProxy demands a configuration object as a constructor parameter (taken from JdkDynamicAopProxy class, authored by Johnson and Hoeller (2015e))

Scanning the *AdvisedSupport's* official documentation shows that it is some kind of configuration object holding essential information about the final proxy. One of the attributes it defines is a list of interfaces (i.e. their corresponding class objects), keeping track of every interface that is implemented by the target bean class (Johnson and Hoeller, 2015b).
This is comprehensible, since the produced proxy has to look like the original bean from the outside. Other beans which depend on a proxied bean must not know they are actually referencing a surrogate object.
Another interesting point is the code that is executed as soon as *getProxy* is called on the JdkDynamicAopProxy class. Listing 3.5 sums up what actually happens:

```
1 @Override
2 public Object getProxy(ClassLoader classLoader) {
3   // log essential information
4   Class<?>[] proxiedInterfaces = AopProxyUtils.
       completeProxiedInterfaces(this.advised);
5   findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
6   return Proxy.newProxyInstance(classLoader,
       proxiedInterfaces, this);
7 }
```

Listing 3.5: Details of the getProxy method (taken from JdkDynamicAopProxy class, authored by Johnson and Hoeller (2015e))

This is where the proxy creation is carried into execution. The AdvisedSupport configuration object is used to bring a wrapped class' interfaces together, which are handed over to the *newProxyInstance* factory method. The statement which is of particular importance when going ahead the *Watson* debugging tool is the *completeProxiedInterface* method residing in the *AopProxyUtils* class. As the semantics of the method denomination suggests, there is more to it than simply collecting a bean's interfaces. According to the official documentation provided by Johnson and Hoeller (2015d), this method also adds additional interfaces to the resulting proxy that are not implemented by the original bean. It says that proxies are always decorated with the *Advised* interface unless the configuration explicitly prohibits this (Johnson and Hoeller, 2015a).
Listing 3.6 shows the relevant lines of code. The *proxiedInterfaces* array receives a reference on the Advised class object and gets processed afterwards during the surrogate's instantiation (Johnson and Hoeller, 2015d).

```
1  public static Class<?>[] completeProxiedInterfaces(
       AdvisedSupport advised) {
2    // omitting the previous lines of code
3    if (addAdvised) {
4      proxiedInterfaces[proxiedInterfaces.length - 1] =
5          Advised.class;
6    }
7    return proxiedInterfaces;
8  }
```

Listing 3.6: Decorating proxies with the Advised interface (taken from AopProxyUtils class, authored by Johnson and Hoeller (2015d))

Although this unimposing interface seems marginal, it finally represents the missing handle on AOP proxies that supplies access to their registered advisors. The following Listing shows a couple of methods which can be employed for advisor management. Further method declarations are skipped for the sake of clarity. For information on the omitted methods as well as the Advised interface in general consider the official documentation (Johnson and Hoeller, 2015a).

```
1  public interface Advised extends TargetClassAware {
2    Advisor[] getAdvisors();
3    void addAdvisor(Advisor advisor) throws AopConfigException;
4    boolean removeAdvisor(Advisor advisor);
5    boolean replaceAdvisor(Advisor a, Advisor b) throws
       AopConfigException;
6  }
```

Listing 3.7: An excerpt of Spring's Advised interface (Johnson and Hoeller, 2015a)

With the knowledge that every AOP proxy also implements the Advised interface, it seems obvious to perform a cast on a JdkDynamicAopProxy or CglibAopProxy

instance and uncover its capabilities for advisor management (Pivotal Software Inc., 2015a). This is one of the core principles *Watson's* functionality is made up of.

How *Watson* fetches AOP proxies

Watson defines a dedicated service class for requesting and retrieving proxied beans from the underlying Spring application context. Listing 3.8 exemplifies the logic which enables the debugging tool to operate on a context:

```
1 public <T> Advised getAdvisedBean(Class<T> aClass) {
2   T bean = springContextProvider.getBean(aClass);
3   Advised advisedBean = null;
4   try {
5    advisedBean = (Advised) bean;
6   }
7   // skipping exception handling ...
8   return advisedBean;
9 }
```

Listing 3.8: Convenience procedure for fetching AOP proxies from an application context (taken from *Watson's* SpringAdvisedBeanService)

Watson's SpringAdvisedBeanService performs the cast described in the previous paragraph. Since the service can not guarantee that the bean the context is asked for actually is an AOP proxy, because it could also be an ordinary Spring bean, the cast is executed regardless of the outcome. If it fails with a *ClassCastException*, the requested bean is not wrapped by a surrogate. Otherwise, if everything works fine and no exception is thrown, the query indeed returned a proxy object.
The *SpringContextProvider* bean referenced inside the method is a convenience class *Watson* declares for accessing the underlying webapp's Spring context as well as the beans it contains. It implements Spring's *ApplicationContextAware* interface that makes the provider class concretize a certain callback method. By means of this method, the Spring container submits it a reference on the underlying application context (Johnson, Hoeller, and Beams, 2015).

```
1 @Override
2 public void setApplicationContext(ApplicationContext
     applicationContext) throws BeansException {
3   appContext = applicationContext;
4 }
```

Listing 3.9: Receiving a reference on the application context from the container (taken from *Watson's SpringContextProvider*)

The method's task is straightforward, since it simply assigns the context reference to a local variable.

3.3.2 Using advisors to change proxy behavior

Having understood how to use the Advised interface to gain access to a proxy's advisors, the next step is to examine the structure of advisors themselves. This helps to establish a basis for the definition of custom implementations that can be employed for instrumentation purposes.

Setting up NameMatchMethodPointcutAdvisor

Spring already comes with several convenience Advisor implementations requiring just a few lines to get an aspect up and running. One of these is the *NameMatchMethodPointcutAdvisor* class (Hoeller and Harrop, 2015b). It already defines a pointcut that can be parametrized with a method name and takes an implementation of the *Advice* interface (Johnson, 2004) as a constructor parameter (Listing 3.10). This interface poses the Spring analogy to the corresponding AOP component. Henceforward, the advisor can be registered at an AOP proxy and is ready for execution.

```
1  public class NameMatchMethodPointcutAdvisor extends
       AbstractGenericPointcutAdvisor {
2    private final NameMatchMethodPointcut pointcut = new
       NameMatchMethodPointcut();
3
4    public NameMatchMethodPointcutAdvisor(Advice advice) {
5      setAdvice(advice);
6    }
7
8    public void setMappedName(String mappedName) {
9      this.pointcut.setMappedName(mappedName);
10   }
11   // skipping the remaining methods ...
12 }
```

Listing 3.10: How to setup a NameMatchMethodPointcutAdvisor (authored by Hoeller and Harrop (2015b))

Watson uses the *MethodInterceptor* interface as a starting point for the interception logic, since it transitively extends Advice and declares a routine that can be used for explicitly breaking into method calls, operating as an around advice (AOP Alliance, 2015a). This is quite enough for enriching business methods with logging, either at the beginning or at the end of their body.

How to integrate code for establishing a peformance measurement is betokened by the following lines of code, defined in *Watson's PerformanceMeasurementInterceptor* class. This class represents an exemplary MethodInterceptor implementation that is already part of the project.

```
1 @Override
2 public Object invoke(MethodInvocation invocation) throws
      Throwable {
3   LoggerProvider.logger.debug("Execution of method " +
       invocation.getMethod().getName() + " started at: " +
4                 LocalDateTime.now());
5   Object result = invocation.proceed();
6   // inject further logging ...
7   return result;
8 }
```

Listing 3.11: Establishing performance logging (taken from *Watson's* PerformanceMeasurementInterceptor)

The interceptor gets passed a *MethodInvocation* object, holding information about the method that was just interrupted (AOP Alliance, 2015b). Whatever a interceptor exactly is supposed to do, it has to consider two important points in any case:

- The MethodInvocation object defines a *proceed* method, giving back control to the proxy which is responsible for the execution of advisors as well as the actual target method (Johnson, Hoeller, and Colyer, 2015). Its invocation is mandatory, since forgetting or neglecting it will result in an everlasting blocking thread.
- As soon as the intercepted method returns, its result (if present) has to be returned by the interceptor. Otherwise the return value would be lost.

Further information and details on interceptor configuration is provided by Pivotal Software Inc. (2015a).
Finally, a complete definition of a NameMatchMethodPointcutAdvisor containing an advice as well as a pointcut could look like this:

```
1 public static NameMatchMethodPointcutAdvisor
     buildPointcutAdvisor(Method method, MethodInterceptor
     interceptor) {
2    NameMatchMethodPointcutAdvisor advisor = new
        NameMatchMethodPointcutAdvisor();
3    advisor.setMappedName(method.getName());
4    advisor.setAdvice(interceptor);
5    return advisor;
6 }
```

Listing 3.12: How to instantiate and setup a custom advisor (taken from *Watson's* WatsonAdvisorBuilder)

3.3.3 Bringing advisors and AOP proxies together

What is still missing is a class that helps *Watson* to manage the configuration of AOP proxies as well as their advisors in a clean and and well-arranged way. This is left to the *AopProxyInstrumentationService* class, which resumes the aggregation of these components in order to get the desired functionality. The code itself is not very complex. Amongst other things, it takes advantage of Java's Reflection API (Oracle Corporation, 2015i) in order to fetch information about bean classes and their methods.
The AopProxyInstrumentationService class defines several methods for adding and removing additional aspects to existing proxies. Since stepping through the implementation of all these procedures would be to verbose, this thesis is content with presenting the code that creates an advisor and appends it on an AOP proxy (Listing 3.13).

```
1  public void configureBeanInstrumentation(String className,
       String methodName, MethodInterceptor interceptor) {
2   try {
3      Class<?> beanClass = Class.forName(className);
4      Advised advisedBean = advisedBeanService.getAdvisedBean(
         beanClass);
5      for (Method method : beanClass.getDeclaredMethods()) {
6        if (method.getName().equalsIgnoreCase(methodName)) {
7          advisedBean.addAdvisor(SpringAdvisorBuilder.
             buildPointcutAdvisor(method, interceptor));
8        }
9      }
10   } // skipping excepion handling ...
11  }
```

Listing 3.13: Creating and appending advisors to proxies (taken from *Watson's* AopProxyInstrumentationService)

For more information on this class this document once more relegates to the *Watson* repository hosted on *Github*.

3.3.4 *Watson's* extension mechanism

A user must be able to extend *Watson* and customize it appropriate to his needs by creating his own interception classes. In order to make this as easy as possible, the tool uses a sophisticated mechanism that saves the user from having to interact with any of its classes to register his own interceptors.

How to plug in custom MethodInterceptors

Custom interceptor implementations have to meet two important requirements in order to integrate properly into *Watson's* extension mechanism:

- As depicted by the PerformanceMeasurementInterceptor mentioned above, it has to implement the MethodInterceptor interface and define a concrete elaboration of its *invoke* method.
- Furthermore, the custom class has to be furnished with the *Watson's @WatsonInterceptor* annotation. The reason for this is the tool's MethodInterceptor auto-discovery policy. How this works is discussed in the subsequent section.

Listing 3.18 gives a brief example, illustrating how a possible implementation might look like:

```
1 @WatsonInterceptor
2 public class MyInterceptor implements MethodInterceptor {
3   @Override
4   public Object invoke(MethodInvoaction invocation) {
5     // do something interesting ...
6   }
7 }
```

Listing 3.14: A custom MethodInterceptor

Exploring *Watson's* MethodInterceptor auto-discovery

The @WatsonInterceptor annotation enables *Watson* to scan every class file available in the class path and register all MethodInterceptors for later use.
For that purpose, *Watson* integrates the *ASM bytecode engineering library* (OW2 Consortium, 2015).
While iterating on the class path locations, an *InputStream* is opened on every class file available (Oracle Corporation, 2015d). On behalf of these InputStream objects, ASM's *ClassVisitor* base class (Bruneton, 2014) can examine different characteristics of the referenced classes' bytecode, like attributes, inner classes or annotations (OW2 Consortium, 2015).
Watson employs these ASM capabilities for checking if a class definition contains the @WatsonInterceptor annotation on the class level. If its presence could be proved, the debugging tool creates and registers an instance of the detected interceptor class, so that it can be selected and applied afterwards. A *HashMap* (Oracle Corporation, 2015c) is used as a storage for MethodInterceptor objects. It caches these instances, avoiding that a certain interceptor that has already been found and instantiated has to go through the process of creation again.
Listing 3.15 sums up the fundamental lines of code to illustrate the auto-discovery's principle of operation. This approach is heavliy inspired by a *JAXenter* article written by Arno Haase (2015).

```
1  @Override
2  public AnnotationVisitor visitAnnotation(String desc, boolean
       visible) {
3   if (desc.equals("Lcom/big/watson/annotation/
       WatsonInterceptor;")) {
4     try {
5       if (!interceptorMap.containsKey(visitedClassName)) {
6         Class<?> interceptorClass = this.getClass().
              getClassLoader().loadClass(visitedClassName);
7         try {
8          interceptorMap.put(visitedClassName,
9          (MethodInterceptor) interceptorClass.newInstance());
10        }
11        // skipping exception handling
12      }
13      methodInterceptors.add(interceptorMap
14          .get(visitedClassName));
15    }
16    // skipping outer exception handling
17    return null;
18 }
```

Listing 3.15: *Watson's* interceptor auto-discovery principle (inspired by Haase (2015))

Although a detailed analysis of ASM and its skills is beyond the scope of this thesis, it's important to notice how it lightens individual adaptions by sparing the underlying JVM's resources at the same time. Since ASM resigns to load classes into the JVM and rather makes use of InputStreams to perform its inspections, the loading process can be triggered lazily as soon as the check for *Watson's* marker annotation has been successful. Otherwise a webapp may suffer from a bloating JVM used to capacity by plenty of classes which are not actually needed. This advantage makes ASM the preferred solution for this task, compared to alternative bytecode engineering libraries like Javassist (Haase, 2015).

3.3.5 Enforcing initial proxy creation

So far, *Watson's* approach of proxy configuration was based on the assumption that beans designated for instrumentation are already wrapped by a surrogate object when fetched from the application context. However, stepping through Spring's proxy creation code reveals that this can only be guaranteed if the framework is able to find an aspect whose pointcut criteria can be applied to a certain bean. Otherwise, there is no need for Spring AOP to make an effort in order to get a bean proxied.
Of course *Watson* could speculate about at least one business method of each class being annotated with *@Transactional* anyway. This would be enough for Spring AOP to recognize that a proxy is required. But apart from considering that there's no guarantee for every bean to make use of such an annotation, *Watson* should not depend on assumptions about the code of the underlying webapp.

It therefore defines a so-called *AopProxyInitializer* aspect as well as another *Watson* marker annotation, in order to ensure that every bean which is intended to be instrumented actually gets fronted by an AOP proxy (Listing 3.16).

```
1  @Aspect
2  public class AopProxyInitializer {
3
4   @Pointcut("execution(public * *(..))")
5   public void publicMethod() {
6   }
7
8   @Pointcut("@within(com.big.watson.annotation.
9         WatsonManaged)")
10  public void watsonManaged() {
11  }
12 }
```

Listing 3.16: *Watson's* AopProxyInitializer aspect

The AopProxyInitializer comprises every class that is annotated with *@WatsonManaged* and defines at least one public method, regardless of its parameters or return type. Since according to Oracle Corporation (2015m), every Java class directly or transitively inherits methods like *toString* from the *Object* class, there is no class being not considered by the latter criterion.
As an alternative, the marker annotation could have been left out and only public methods could have been considered a proxification criterion. What seems to cause less effort in the first place produces a serious issue. Since every class is affected, as explained above, all framework classes get fronted by surrogates, too. This causes a Spring application to crash because some of the framework's classes do not accept to get injected a proxy.

3.4 Introducing remote capabilities with Spring JMX

For arranging the communication with *Watson* and being able to use it from a local or remote host, the underlying application has to be JMX-enabled. As seen in section 2.7.2, this demands the registration several beans that come with the Spring JMX project (Walls, 2014, pp. 524-527).

3.4.1 Registration of required Spring JMX beans

Registering a Spring MBean server as well as a Spring MBean exporter is neither very complex nor verbose. *Watson* ships with its own configuration class that already includes all necessary bean definitions for enabling JMX in the context of a webapp that integrates the tool without further ado (Walls, 2014, pp. 524-527):

```
1  @Bean
2  public MBeanServer mBeanServer() {
3    MBeanServerFactoryBean serverFactoryBean = new
        MBeanServerFactoryBean();
4    serverFactoryBean.setLocateExistingServerIfPossible(true);
5    serverFactoryBean.afterPropertiesSet();
6    return serverFactoryBean.getObject();
7  }
8
9  @Bean
10 public MBeanExporter mBeanExporter(WatsonControllerMBean
     controllerMBean) {
11   MBeanExporter mBeanExporter = new MBeanExporter();
12   Map<String, Object> beans = new HashMap<>();
13   beans.put(watsonControllerMBean.CONTROLLER_MBEAN_NAME,
        controllerMBean);
14   mBeanExporter.setBeans(beans);
15   return mBeanExporter;
16 }
```

Listing 3.17: *Watson* provides a MBean server as well as a MBean exporter bean (taken from WatsonConfig)

The *MBeanServerFactoryBean* checks if the JVM has already started a MBean server. If no running MBean server is detected, it takes care of creating and starting a new one (Walls, 2014, p. 527).
The *MBeanExporter* bean holds a simple *Map* (Oracle Corporation, 2015k) that contains references on one or more ordinary Spring beans. The exporter announces these beans to the MBean server, which finally publishes them as MBeans (Walls, 2014, pp. 524-525). *Watson* adds a special bean named *WatsonControllerMBean*, which serves as the remote interface for controlling the debugging tool on behalf of JConsole.

3.4.2 *Watson's* remote interface

The WatsonControllerMBean supplies several methods for configuring the debugging tool as soon as it has been started along with an application server. These methods cover the following functionality:

- *showAvailableAspects*: This operation triggers the class path scan described in section 3.3.4 and returns a list containing every MethodInterceptor implementation that can be found.
- *listAspectsForBean*: Taking a bean's fully qualified name as a parameter, this procedure lists every aspect that is currently registered on a certain Spring bean. Theoretically, a bean can be intercepted by more than one aspect at the same time. The numeration ignores aspects that were created and added by the Spring Framework itself, like the ones defining transactional behavior.
- *selectCurrentAspectType*: With this method, a user can choose one of the available MethodInterceptors for being applied during the next bean instrumentation process.
- *prepareBean*: This method expects to be handed over a bean's fully qualified class name as well as the name of a method the class defines. The currently selected MethodInterceptor will be used to build an advisor object, which will be added to the proxy subsequently.
- *removeAspectsFromBean*: In order to cancel the instrumentation of a Spring bean, this procedure takes the respective class' fully qualified name as well as an index that has to point to an advisor residing in the list of the proxy. Before actually executing the removal of an advsior, *Watson* filters aspects added by the framework itself. Thus, it is not possible for a user to accidentally eliminate e.g. a bean's transactional behavior.

3.4.3 Configuring JMX security

In order to avoid additional complexity in the first instance, *Watson* shall be accessible via JMX without further restrictions, e.g. in the form of authentication or auhtorization. It turned out that the default JMX settings fit these requirements, so there are no further adaptions that have to be applied.
Whether any of the JMX security mechanisms presented in section 2.7.3 should be put into operation is left to the user or his developer team.

3.5 How to setup a webapp for *Watson*

Similar to *Watson's* customization principle, its integration into an existing Spring-based webapp is aimed at convenience. Launching the debugging tool along with a server application requires only two steps:

- *Watson* has to be present on the application's class path in the form of a JAR file.
- The *WatsonConfig* bean has to be defined in the application's Spring context. It makes no difference whether a webapp relies on Java or XML configuration for that purpose.

```
1 @Bean
2 public WatsonConfig watsonConfig() {
3   return new WatsonConfig();
4 }
```

Listing 3.18: Setting up *Watson* merely requires to instantiate its configuration bean

Since the WatsonConfig bean already includes the definitions of all the other necessary beans, there is nothing left to do to get *Watson* to work. As soon as the webapp and its underlying server have been started, the WatsonCotrollerMBean appears when launching JConsole and connecting it to the application.

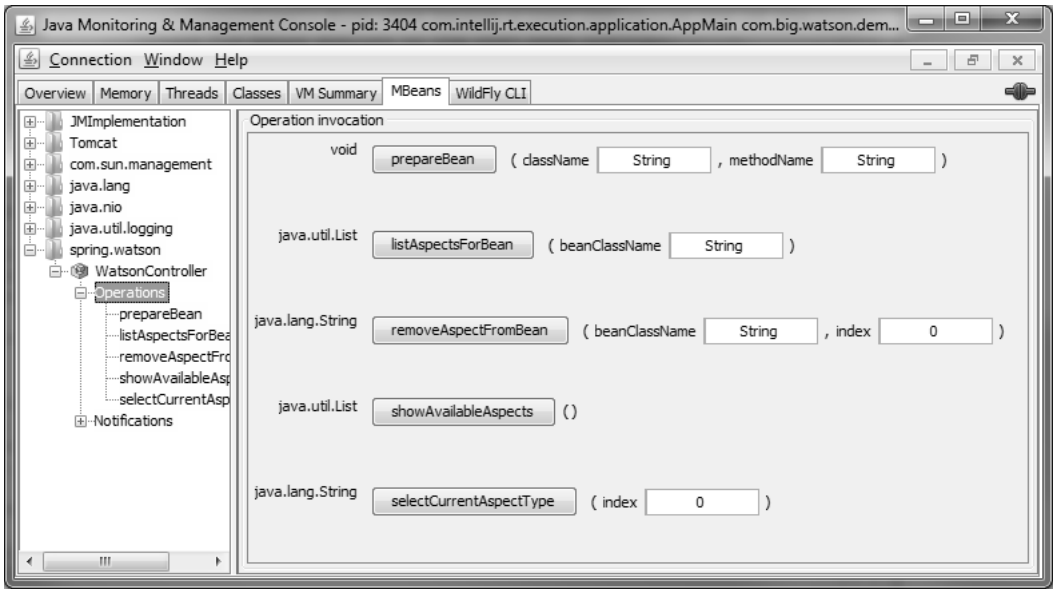

Figure 3.2: Inspecting *Watson's* MBean with JConsole

In case a user wants to make himself familiar with *Watson* before integrating it into an existing webapp, the tool comes with an embedded *Spring Boot* application which can be started instantly. Launching it only requires to run the *SampleController's* main method. This example is based on a tutorial provided by Pivotal Software Inc. (2015f).
For further information on the Spring Boot project consider Walls (2014, pp. 540-569) as well as the official documentation (Pivotal Software Inc., 2015f).

Chapter 4

Implementing *SherLog* with the Instrumentation API and Javassist

The fourth chapter of this thesis concentrates on *SherLog*, a debugging tool identical to *Watson* as far as its function is concerned. In contrast to *Watson*, it is not limited to only Spring-based applications. In fact, *SherLog*'s purpose is to not depend on any Java enterprise environment, rather being applicable to each of them. To achieve this level of conformity, the tool must not depend on platform dependent projects and frameworks like Spring AOP. Instead, it relies on the Instrumentation API and Javassist for the insertion of additional logging. Javassist as a third party library can be used in any Java application, whereas the Instrumentation API is actually part of Java SE.
As for the development's frame conditions, this debugging tool was implemented and compiled with JDK 8, as well as tested against a *WildFly Application Server (WildFly AS) 8.2.0*.
The following remarks will come along with several code snippets taken from the *SherLog* project, whose source code has been published under `https://github.com/Patrick-Kleindienst/BIG-SherLog`.

4.1 Conception and core functionality

Before immerging into the details of *SherLog's* implementation, its high-level functionality shall be highlighted. Figure 4.1 illustrates the basic concepts and how they integrate into the environment of a webapp deployed on an application server.

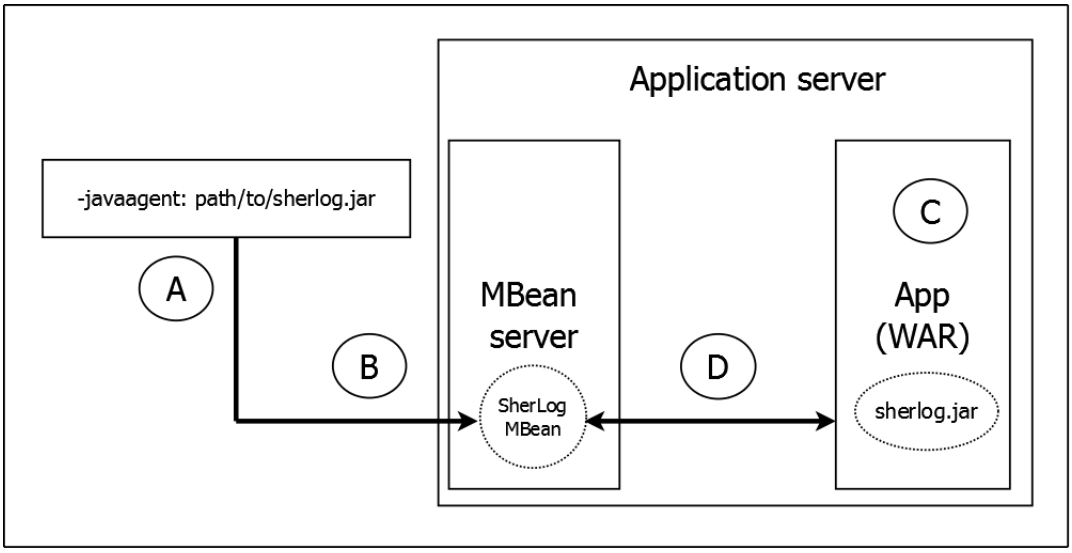

Figure 4.1: *SherLog* registers a MBean which serves as a cutting point for remote communication with the deployed application

SherLog is packed as a JAR and executed as a Java agent [**A**] in order to get hold of an Instrumentation object, which constitutes the agent's primary purpose. This object poses a handle for accessing all webapp classes currently loaded by the JVM (Oracle Corporation, 2015e).
The agent's second main function is to instantiate and register a certain MBean [**B**]. On the one hand, this MBean manages the Instrumentation instance and acts as a bridge between the MBean server and the running application [**D**], providing access to the deployed webapp and enabling a client to directly work on its compiled classes [**C**]. On the other hand, the MBean is published by the platform's MBean server, allowing the execution of operations on the bean via JDK tools like JConsole. The MBean then induces the insertion of suitable logging statements on behalf of the Instrumentation object.
Note that the *sherlog.jar* must also be present in the WAR file's *lib* directory [**C**], since the tool brings its own dedicated *Logger* class, which has to be visible to the class loader that loads the webapp classes (Apache Software Foundation, 2015a). This circumstance is caused by a particular class loading policy that was established by *JBoss* with the seventh major release of the *JBoss Application Server (JBoss AS)* (Red Hat Inc., 2015b). This policy will be covered in the last section of this chapter, when this document deals with how WildFly has to be configured for running *SherLog* properly.

4.2 *SherLog's* implementation details

Instead of rearranging classes and methods by building constructs like aspects in order to gain additional logging capabilities, *SherLog* tackles this concern at the bytecode level. It therefore resorts to the Java Instrumentation API and its Instrumentation interface (Oracle Corporation, 2015e) as well as Javassist (Chiba, 2015) for having not to operate on plain bytecode in a low-level manner and avoiding class loading issues (see section 2.5.1).
The following sections will cover *SherLog's* approach and project architecture in detail.

4.2.1 Fetching an Instrumentation instance

Before *SherLog* is ready to do its work, it needs to ask the JVM for an implementation of the Instrumentation interface. Therefore, it provides a Java agent class including a premain callback method, as shown in Listing 4.1. The agent class receives a reference to the Instrumentation object, which is actually an instance of the InstrumentationImpl class, and stores it in a static variable in order to make it available to other project classes via a getter method.

```
1  public class SherlogSetupAgent {
2
3   private static Instrumentation instrumentation;
4
5   public static void premain(String agentArgs, Instrumentation
        instr) {
6       instrumentation = instr;
7       // omitting further tasks at first
8   }
9  }
```

Listing 4.1: Fetching an Instrumentation object from the JVM

To make the JVM find the *SherlogSetupAgent* class and invoke its premain method, the sherlog.jar contains an approriate MANIFEST.MF file:

```
1  Manifest-Version: 1.0
2  Premain-Class: com.big.sherlog.agent.SherlogSetupAgent
3  Can-Retransform-Classes: true
```

Listing 4.2: *SherLog's* MANIFEST.MF file

This manifest indicates the premain-class and configures the JVM to accept the retransformation of classes, as required for the tool's bytecode modifications. After the execution of the premain method, the Instrumentation object can be accessed and used whereever the tool needs its capabilities.
When compiling and packaging *SherLog* with Maven, it must be ensured that

MANIFEST.MF is placed in the JAR's *META-INF* directory (Oracle Corporation, 2014a). It is also possible to make use of the *Maven Assembly Plugin* (Apache Software Foundation, 2015b) for instructing Maven to generate this file based on prepared attribute values and put it into the correct folder when it packages the JAR.
As it does not play a significant role how exactly the MANIFEST.MF is created, this will not be deepend here. The project's *pom.xml* provides further information on how to automate this task with Maven.

4.2.2 Implementation of ClassFileTransformers

The second chapter already pointed out that the Instrumentation object needs to be equipped with ClassFileTransformer implementations in order to know what kind of changes should be applied to a certain class definition (Oracle Corporation, 2015e). *SherLog* furnishes a couple of such classes that can be used out of the box.

Providing a base ClassFileTransformer

Listing 4.3 gives a rough overview of the abstract *BaseTransformer* class. It implements the ClassFileTransformer interface and represents *SherLog's* starting point for concrete implementations:

```
1  public abstract class BaseTransformer implements
       ClassFileTransformer {
2
3    private BaseCodeIntegrator codeIntegrator;
4    /* omitting further instance members etc. for clarity */
5
6    @Override
7    public byte[] transform(ClassLoader loader,
8                            String classname,
9                            Class<?> classBeingRedefined,
10                           ProtectionDomain protectionDomain,
11                           byte[] classfileBuffer)
12                           throws IllegalClassFormatException
13   {
14     classname = classname.replace("/", ".");
15     if (classname.equals(this.className)
16           && loader.equals(this.classLoader)) {
17         return codeIntegrator.performCodeIntegration
18                (className, methodName, methodSignature,
                    classLoader, classfileBuffer);
19     }
20     return classfileBuffer;
21   }
22 }
```

Listing 4.3: *SherLog's* ClassFileTransformer base implementation

The BaseTransformer's major task is to provide an implementation for the transform callback, since this is required by the contract of the interface (Oracle Corporation, 2015e). When invoking this method, the Instrumentation API hands over several parameters, but not all of them are relevant for *SherLog's* intention. For example, it neglects the *ProtectionDomain* parameter, because it does not define any domain-specific permissions. However, the transform method uses the class loader as well as the class name passed to check if the class, which is about to be modified, is actually the class a client thinks it is. From the JVM's point of view, a class cannot definitely be identified by only its name. Moreover, a class' identity consists of its fully qualified name plus the class loader that loaded it (Liang and Bracha, 1998). The additional verification step ought to avoid any misunderstandings about which class has to be instrumented.
Subsequently, the transform method takes the parameters and transfers them to the *BaseCodeIntegrator* attached to the BaseTransformer class. A BaseCodeIntegrator defines an interface for extracting the bytecode modification concerns from the ClassFileTransformer classes. Each BaseTransformer instance charges a concrete implementation through its provided constructor.
Relying on the abstraction principle in case of the BaseTransformer class leads to several advantages. On the one hand, every concrete ClassFileTransformer inheriting from BaseTransformer saves the need for implementing the transform method on its own. On the other hand, classes that depend on BaseTransformer are decoupled from its subclasses and only have to be aware of the abstract supertype. Thus, the concrete implementation in use can be exchanged dynamically.

Encapsulation of bytecode modification concerns

SherLog introduces the concept of BaseCodeIntegrators in order to define seperate units of code, which hide the complexity of bytecode operations from the ClassFileTransformers. Gathering this code inside of the BaseTransformer class would have been possible, but would also have made the whole project's code less clear and flexible. Figure 4.2 shows a rough UML diagram illustrating this appliance of the *strategy* design pattern (Gamma, Helm, and Johnson, 2001, pp. 373-384).

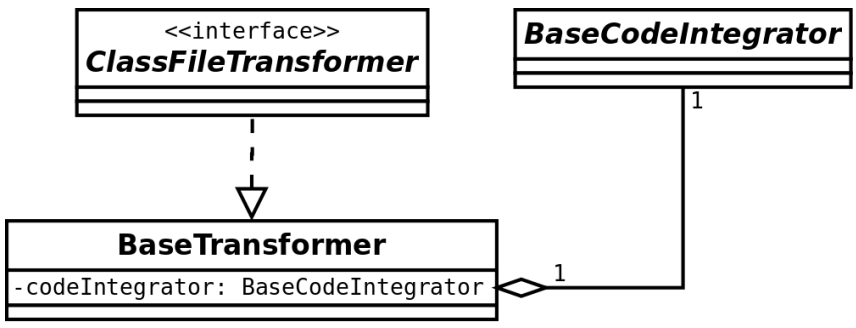

Figure 4.2: *SherLog* uses the strategy pattern in order to provide flexibility

The manner of how these BaseCodeIntegrators are implemented is comparable to the BaseTransformer class hierarchy, prescinding functionality that is shared by several classes in a common base class. It encapsulates code that would otherwise have to be re-written by every subclass and acts as a common interface from dependent classes' point of view.
The BaseCodeIntegrator class contains three methods. One of these is declared abstract, requiring an implementation as soon as a concrete subclass is inferred:

```
1  public abstract class BaseCodeIntegrator {
2
3   protected abstract CtMethod enhanceMethodCode(CtMethod
       ctMethod);
4   // omitting other methods ...
5  }
```

Listing 4.4: The BaseCodeIntegrator class delegates instrumentation decisions to subclasses

The choice of what kind of modification should be applied to the CtMethod object passed as a parameter is left to the implementations of the subclasses. *SherLog* supplies several BaseCodeIntegrator subclasses which provide concrete elaborations of this method, inserting logging code for e.g. performance measurements or parameter monitoring.
Listing 4.5 demonstrates how a solution might look like. This code snippet is taken from the *PerformanceCodeIntegrator* class, which is one of *SherLog's* pre-defined implementations.

```
1  @Override
2  protected CtMethod enhanceMethodCode(CtMethod ctMethod) {
3   try {
4      ctMethod.addLocalVariable("startTime", CtClass.longType);
5      ctMethod.addLocalVariable("stopTime", CtClass.longType);
6      // inserting more code ...
7      ctMethod.insertAfter(LOG_TIME_DIFF);
8      }
9     // omitting exception handling ...
10  return ctMethod;
```

Listing 4.5: Enhancing a method with logging code that documents the execution duration (taken from *SherLog's* PerformanceCodeIntegrator)

The enhancement introduces two local variables of type *long* in order to keep hold of the timestamp when the execution of the instrumented method starts, as well as when it ends. Afterwards, it computes the duration of the execution process and inserts a statement that logs the result into a file. The *LOG_TIME_DIFF* String hides away the correspondent logging expression:

```
1  private static final String LOG_TIME_DIFF = PROVIDED_LOGGER +
       ".debug(\"Execution took \" + java.lang.String.valueOf(
     stopTime - startTime) + \" ms\");";
```

Listing 4.6: A sample logging statement that computes and logs a method's execution duration (taken from *SherLog's* PerformanceCodeIntegrator)

The required logger is accessed through the *PROVIDED_LOGGER* variable. It provides the fully qualified name of *SherLog's* logging class plus the Logger's name. This indication is supposed to enable the webapp's class loader to find and load the *LoggerProvider* class.

```
1  protected static final String PROVIDED_LOGGER =
     LoggerProvider.class.getName() + ".LOGGER";
```

Listing 4.7: Referencing the tool's custom Logger (taken from *SherLog's* BaseCodeIntegrator)

The LoggerProvider class merely defines a log4j Logger and makes it available by the definition of a static variable. The related *log4j.properties* file that contains the Logger configuration will be skipped at this point, since it defines a straightforward *FileAppender* for storing the output in a seperate log file (Guelcue, 2012). More detailed information on the log4j configuration can be found in the project's repository on *Github*.

```
1  public class LoggerProvider {
2   private static final String LOGGER_NAME =
3         "BIG-SherLog-Logger";
4   public static Logger LOGGER = Logger.getLogger(LOGGER_NAME);
5  }
```

Listing 4.8: *SherLog* defines and publishes its own Logger

The two remaining methods residing in the BaseCodeIntegrator class is where the debugging tool applies the subclass-specific modifications, starting with the procedure shown in Listing 4.4. This method can be considered one of the core elements of *SherLog*, using Javassist to retrieve the desired class handle from the ClassPool and inducing the actual instrumentation on the CtClass that may has been found.

```
1  public byte[] performCodeIntegration(String className,
2                                       String methodName,
3                                       String methodSignature,
4                                       ClassLoader loader,
5                                       byte[] bytes)
6  {
7    className = className.replace("/", ".");
8    ClassPool classPool = new ClassPool(true);
9    classPool.appendClassPath(new LoaderClassPath(loader));
10   classPool.appendClassPath(new ByteArrayClassPath(className,
         bytes));
11
12   try {
13      CtClass ctClass = classPool.get(className);
14      if (!searchAndReplaceMethod(ctClass, methodName,
           methodSignature)) {
15        if (methodSignature == null) {
16          LoggerProvider.LOGGER.error("Could not find method
               for name: " + methodName);
17        } else {
18          LoggerProvider.LOGGER.error("Could not find method
               for name: " + methodName + " (Signature: " +
               methodSignature + ")");
19        }
20      }
21     return ctClass.toBytecode();
22   }
23   // omitting exception handling ...
24   return null;
25 }
```

Listing 4.9: The *performCodeIntegration* method searches for a CtClass object in the ClassPool and triggers its instrumentation (taken from *SherLog's* BaseCodeIntegrator)

Since the whole action is rather complicated and embraces lots of different operations, *SherLog* swappes out the method replacement into a private utility method:

```
1  private boolean searchAndReplaceMethod(CtClass ctClass,
       String methodName, String methodSignature) throws
       NotFoundException, CannotCompileException {
2    boolean atLeastOneMethodFound = false;
3    for (CtMethod ctMethod : ctClass.getDeclaredMethods()) {
4      if (ctMethod.getName().equalsIgnoreCase(methodName)) {
5        if (methodSignature != null && ctMethod.getSignature().
             equals(methodSignature)) {
6            ctClass.removeMethod(ctMethod);
7            ctClass.addMethod(enhanceMethodCode(ctMethod));
8            atLeastOneMethodFound = true;
9        } else if (methodSignature == null) {
10           ctClass.removeMethod(ctMethod);
11           ctClass.addMethod(enhanceMethodCode(ctMethod));
12           atLeastOneMethodFound = true;
13         }
14       }
15     }
16     return atLeastOneMethodFound;
17   }
```

Listing 4.10: Finding and replacing an existing method (taken from *SherLog's* BaseCodeIntegrator)

The following enumeration sums up the single steps which are performed when the method seen in Listing 4.9 is called:

- A new ClassPool is instantiated and initialized. This is necessary because having Javassist caching older representations of application classes may have unexpected side effects. A fresh ClassPool guarantees that the method will always work on bytecode that is up-to-date (JBoss Inc., 2015a).
  Furthermore, additional class paths are appended to the ClassPool, enabling Javassist to locate the application classes. This needs to be done because as a part of the Java agent, the ClassPool does not know about the class loader that loads the webapp classes as well as the paths it searches for these (JBoss Inc., 2015a).

- With the CtClass object in hand, the next step is to search for the method that conforms to the given name and the optional signature. If there is more than one method matching the given name and no explicit signature is present, the instrumentation will be applied to every overloaded method whose name matches *methodName*. Notice that a client gets informed through an error logging message in case that no suitable method could be found.

- In order to properly execute the instrumentation, the relevant method is removed from the class at first. Afterwards, its modified representation is reattached. Notice the call to the *enhanceMethodCode* method, that actually implements the insertion of logging code as seen above (Listing 4.4).
- At least, the updated class' bytecode is returned as an array of plain byte values. As already mentioned in section 2.5.1, it's important to avoid trying to reload the class explicitly by calling for example *ctClass.toClass()* (JBoss Inc., 2015a). Instead, the Instrumentation API itself cares for the update of the current class definition (Oracle Corporation, 2015e). Using a new class loader would result in the creation of a completely new class existing side by side with the original class (Liang and Bracha, 1998). However, it is essential that only the existing class definition is modified, since *SherLog's* purpose is to also affect instances that have already been created from the classes scheduled for instrumentation.

Adding custom ClassFileTransformes to *SherLog*

The purpose of *SherLog's* code design goes beyond readability and loose coupling. It is also supposed to guarantee that a client is able to extend the tool's logging capabilities in a simple manner. To make use of this extensibility, he has to implement the following components:

- A custom subclass of the abstract BaseCodeIntegrator class
- A custom subclass of the abstract BaseTransformer class, annotated with *@SherlogTransformer*

An exemplary custom BaseTransformer subclass is shown in Listing 4.11. Since the complexity is hidden from concrete implementations, the class definition itself requires just a few lines of code:

```
1 @SherlogTransformer
2 public class MyCustomClassFileTransformer extends
      BaseTransformer {
3
4   public MyCustomClassFileTransformer() {
5     super(new MyCustomCodeIntegrator);
6   }
7 }
```

Listing 4.11: A custom ClassFileTransformer

The thesis skips an example of a custom BaseCodeIntegrator, since this has already been covered in the previous section.

ClassFileTransformer auto-discovery via class path scan

The document already gave an overview of using ASM for introducing custom interceptors to the *Watson* debugging tool, as seen in section 3.3.4.
SherLog can benefit from the same principle, offering a possibility to be extensible in a simple manner. The only difference here is that *SherLog* defines its own *@SherlogTransformer* annotation for identifying and registering every *BaseTransformer* subclass available on the class path.
Listing 4.12 shows the *SherLog* version for the sake of completeness. Doing it in exactly the same way as *Watson*, the notion of combining a class path scan with ASM's lazy class loading capabilities is once again based on the *JAXenter* article authored by Haase (2015) .

```
1  @Override
2  public AnnotationVisitor visitAnnotation(String desc, boolean
       visible) {
3    if (desc.equals(TRANSFORMER_ANNOTATION)) {
4     try {
5      this.getClass().getClassLoader().loadClass(className);
6     }
7     // omitting exception handling ...
8  }
```

Listing 4.12: Implementing ClassFileTransformer auto-discovery with ASM (taken from *SherLog's* TransformerClassVisitor, inspired by Haase (2015))

4.2.3 Defining an InstrumentationService

For managing the instrumentation operations that shall be executable remotely, *SherLog* defines a particular *InstrumentationService* class. This class encapsulates the complexity of configuring the Instrumentation object as well as its ClassFileTransformers. It provides several convenience methods, enabling the tool's JMX components to control the instrumentation of application classes in an easy fashion.

Managing ClassFileTransformers

Listing 4.13 illustrates how the InstrumentationService class handles adding and removing BaseTransformer instances:

```
1  private void instrument(Class aClass, String methodName,
       String methodSignature) {
2    if (instrumentation != null) {
3      initializeTransformer(aClass.getClassLoader(), aClass.
         getName(), methodName,  methodSignature);
4      instrumentation.addTransformer(transformer, true);
5      try {
6        instrumentation.retransformClasses(aClass);
7      } catch (UnmodifiableClassException e) {
8        e.printStackTrace();
9      } finally {
10       instrumentation.removeTransformer(transformer);
11     }
12   } else {
13     throw new RuntimeInstrumentationException(
14     "Instrumentation value must not  be null!");
15   }
16 }
```

Listing 4.13: Encapsulation of the Instrumentation object's setup (taken from *SherLog's* InstrumentationService)

Note that each BaseTransformer object has to be initialized before it can be attached. As already seen in section 4.2.2, it has to gain all the information that is necessary to indentify the correct class as well as the desired method without a doubt.
It is also important that any BaseTransformer is removed from the Instrumentation instance immediatley, after its changes have been applied. In this way, it can be ensured that a client does not get any unexpected results, since the impact of ClassFileTransformers is additive if several of them are registered at the same time. By removing a them instantly, they can be prevented from remaining in the Instrumentation object's list of ClassFileTransformers and causing the unintended integration of instrumentation code that is no longer scheduled for appliance (Oracle Corporation, 2015e).
If a user gets the possibility to perform changes on class definitions, he also needs a converse operation in order to remove them if they are not desired any more. For this purpose, *SherLog* utilizes the fact that every transformation starts with the original class bytes (Oracle Corporation, 2015e). So, if a modified class should be reset to its original state, the logical approach is to trigger another transformation, but entirely ignoring the registration of any ClassFileTransformer (Listing 4.14). As a result, this kind of transformation makes a class adopting the state it occupied when it has initially been loaded.

```
1 public void restoreClass(Class aClass) {
2   try {
3     instrumentation.retransformClasses(aClass);
4   } catch (UnmodifiableClassException e) {
5     e.printStackTrace();
6   }
7 }
```

Listing 4.14: Resetting a class definition to its original state (taken from *SherLog's* InstrumentationService)

4.3 Introducing remoteness with JMX

For enabling clients to communicate with *SherLog* from the local host as well as from a remote machine, this tool also relies on the JMX technology. The configuration is very similar to what has already been shown when having dealt with *Watson's* JMX setup (see section 3.4).

4.3.1 Defining the *SherLog* MBean operations

The implementation of the *SherLog* MBean follows the standard pattern as demonstrated in section 2.7.1. Its intention is to enable a user to manage an application with minimal knowledge about the underlying project's structure and the naming conventions of pertaining classes. The following operations defined in the *JmxInstrumentationServiceMBean* class ought to guide the user through the instrumentation process:

- *setBasePackage*: A user can define an optional base package that acts as a filter and delimits the application's domain that is considered when asking e.g. for a list of loaded classes. Specifying a base package is highly recommended, since the unfiltered list of classes is not restricted to application classes, but also includes a large quantity of e.g. JDK and server bootstrapping classes.

- *listLoadedClassesByBasePackage*: This method returns the name of every class that is currently loaded by the JVM, respecting the optional base package.

- *listMethodNamesForClass*: This is a convenience method that accepts the fully qualified name of a loaded class as a parameter and returns every method a class implements, including private or protected ones.

- *listSignaturesForMethod*: When passed the fully qualified name of a class and the name of a method it includes, this method delivers every signature available for methods matching this name. This operation is helpful if a class defines overloaded methods, but only a particular one shall be instrumented.

- *instrumentMethod*: Triggering the insertion of additional logging can be done with this method. It takes a class name, a method name as well as an optional signature as parameters.
- *resetClassTransformation*: With this operation, a class can be reset to its original state. It must be given the fully qualified class name.
- *listAvailableTransformations*: This method returns a list of all ClassFileTransformers available in the class path. For gathering the implementations, it relies on a class path scan, as seen in section 4.2.2.
- *selectTransformer*: The available ClassFileTransformers are stored in a list and can be selected by their indices. The tool ensures that only one ClassFileTransformer can be applied at the same time.

4.3.2 Registration of the *SherLog* MBean

Instantiating and registering *SherLog's* MBean has been skipped when the Java agent's premain method was presented in Listing 4.1. Listing 4.15 shows the missing steps in the context of the Instrumentation setup as discussed in section 4.2.1:

```
1  public static void premain(String agentArgs, Instrumentation
       instr) {
2    instrumentation = instr;
3    ObjectName objectName = new ObjectName("com.big.sherlog:
       type=SherlogService");
4    mBeanServer.registerMBean(new JmxInstrumentationService(
       instrumentation), objectName);
5    // skipping exception handling
6   }
```

Listing 4.15: Registering the *SherLog* MBean (taken from *SherLog's* SherlogSetupAgent)

The setup shown above is sufficient for the *SherLog* MBean to be registered and published properly. Its operations and attributes are now accessible through an appropriate JMX monitoring tool like JConsole.

4.3.3 Configuring JMX security

As already explained in the connection with the *Watson* debugging tool, this thesis disclaims additional JMX security settings (see section 3.4.3). However, a company or developer team is free to determine the level of JMX security according to its individual needs.

4.4 Setting up WildFly 8 for *SherLog*

Having implemented the logging insertion services as well as having registered a MBean for remote access, *SherLog* can now be set in action. However, trying to start it along with the server fails in the first instance. This section steps through the configuration adjustments required to make the WildFly AS 8 server execute the *SherLog* agent as expected and start properly.

4.4.1 Preparing JBoss WildFly for instrumentation

Trying to install *SherLog* with the -javaagent option when launching the WildFly server fails with an *IllegalStateException*, accompanied by the following description:

The LogManager was not properly installed
(you must set the "java.util.logging.manager" system property to
"org.jboss.logmanager.LogManager")

What causes this behavior is the JBoss modularization concept (Red Hat Inc., 2015c). The next paragraph will cover this in short.

The JBoss modularization concept

An innovation that came up with JBoss AS 7 was the concept of modularization. Henceforward, the server's infrastructure was split up in several units, so-called *modules*. These modules are strictly isolated from each other, but are still able to declare dependencies to one or more other modules. in this way, a module can take advantage of another one's capabilities. The logging module can be used as an example. A component that wants to use logging defines a dependency to the logging module and gains the ability for using the module's Logger classes (Red Hat Inc., 2015c).
A module that is intended to be available to others must be loaded and initialized at first, in order to be ready for use. However, since a Java agent is executed right before the server bootstrapping process is launched, it may happen that the agent requires some dependencies which are not available yet. This is why trying to start the WildFly server along with the agent causes an IllegalStateException as mentioned above (Red Hat Inc., 2015c).
This problem can be resolved by configuring the *LogManager* in *%JBOSS_HOME%/bin/**standalone.conf.bat***. Several settings must be applied, following the instructions given by the exception's stack trace:

- Enabling the JBoss LogManager by assigning it to the *java.util.logging.manager* system parameter
- Making the LogManager visible to every class loader by setting the *jboss.modules.system.pkgs* JVM parameter

- Attaching the JAR that includes the LogManager classes to the system class loader's class path

The name of the configuration file as well as the syntax of system and environment variables depend on the underlying operating system. Listing 4.16 shows the respective entries and how they have to be added to the batch file in a *Microsoft Windows* environment:

```
1 set JAVA_OPTS=%JAVA_OPTS% -Djava.util.logging.manager=org.
     jboss.logmanager.LogManager"
2 set "JAVA_OPTS=%JAVA_OPTS% -Xbootclasspath/p:%JBOSS_JOME%/
     modules/system/layers/base/org/jboss/logmanager/main/jboss
     -logmanager-1.5.2.Final.jar"
3 set "JAVA_OPTS=%JAVA_OPTS% -Djboss.modules.system.pkgs=org.
    jboss.logmanager"
```

Listing 4.16: Configuring the JBoss LogManager in *Microsoft Windows 7*

4.4.2 JBoss WildFly class loading policy

With the *LogManager* configured, the WildFly server starts properly. However, invoking *SherLog's* instructions now raises a *VerifyError*. The reason for this error is that an application which should be instrumented depends on *SherLog's* LoggerProvider class. This introduces difficulties concerning the class loading mechanism of the WildFly server.

How classes are usually loaded

The JBoss WildFly application server's class loading mechanism is quite different from the common hierarchical class loading pattern described by Liang and Bracha (1998). According to their paper, each class loader, except for the *system class loader*, holds a reference to a parent class loader. As soon as a class loader is asked to load class, it primarily delegates the request to its parent loader, following a parent-first policy. In this way, a class loading request may climb up the class loader chain until the top-level system class loader is reached. A class loader is allowed to try to load a class only if the parent chain did not succeed (Liang and Bracha, 1998).

How a JBoss application server deals with class loading

In a JBoss AS environment, deployments are also treated as modules, which are loaded and isolated by dedicated *ModuleClassLoaders* (Bailey, Lloyd, and Diesler, 2014). Following this approach, a JBoss application server can guarantee that multiple webapps can operate seamlessly on the same server instance at the same time (Red Hat Inc., 2015b).

When passing the sherlog.jar as a Java agent, its classes are loaded by an instance of *AppClassLoader*, which is responsible for bootstrapping the server process and acts as the system class loader, too. This class loader simultaneously represents the parent loader of all ModuleClassLoader instances (Red Hat Inc., 2015b). Taking the JVM's familiar class loading mechanism as given results in the assumption that if the webapp needs to load a class from sherlog.jar, it could simply delegate this task to its parent loader, which exactly is the AppClassLoader. But since the *SherLog* Java agent is not part of the deployed webapp, the deployment's ModuleClassLoader not seems to be allowed to load classes residing in sherlog.jar. The official JBoss documentation does not cover this in detail, but it can be assumed that this behavior may be the consequence of switching the standard parent-first loading policy to a child-first policy. This would mean that each ModuleClassLoader can try to load a class in the first place and fails immediately if a class is not available or if the class' origin is not considered trustworthy.
Remembering the JBoss modularization concept described in section 4.4.1, the ModuleClassLoader's behavior makes quite sense. The *SherLog* agent can be considered being executed apart from a the instrumented webapp, but the webapp itself does not declare an appropriate dependency to the sherlog.jar as demanded by the module mechanism's architecture. Instead, it operates isolated from every module that does not excplicitly declare a dependency on it and vice versa (Red Hat Inc., 2015b).
Going beyond isolation, security may also be a reason for the strict separation of modules. Applying a child-first class loading policy facilitates control of which classes can be loaded into a running application. Thus, a server can protect inherent running webapps by establishing a barrier against the infiltration of malicious code.

Satisfying the ModuleClassLoader

Enabling a ModuleClassLoader to find *SherLog's* LoggerProvider class can be achieved in two ways. The first possibility is to follow the JBoss modularization approach by defining a discrete *SherLog* module, which contains the sherlog.jar. However, defining a module for a JBoss application server is not that trivial although the official documentation provides an appropriate manual. Moreover, a module requires several XML files for installing it in the server and for declaring its dependencies (Red Hat Inc., 2015b).
Therefore, this thesis chooses another way and simply adds the sherlog.jar to the webapp's WAR artifact before deploying it on a server, as already shown in figure 4.1 at the beginning of the chapter. As a consequence, the ModuleClassLoader which loads the application classes has no difficulties to find the LoggerProvider class, since it is packaged as part of the webapp itself. Setting up *SherLog* this way saves the need for a bunch of XML configuration and can be applied much faster.

Of course it is arguable that using a Logger which has already been defined and initialzied by an application class instead of *SherLog's* Logger saves the effort to deal with an application server's class loading behavior. This might be true considering the case that a webapp can guarantee to provide a Logger as well as a suitable configuration. But since one of *SherLog's* goals is to integrate seamlessly into different applications, it should not rely on the underlying deployment's settings.

4.4.3 Applying *SherLog's* log4j configuration

Despite having satisfied every module dependency as well as having bypassed all class loading issues, logging with *SherLog's* predefined Logger residing in the LoggerProvider class still does not work.
The reason is that the WildFly server already includes a logging module as well as a corresponding configuration. Hence, the logging settings of deployment modules are ignored by default. Though, WildFly offers the possibility to prevent this configuration from being applied to deployments. This requires an additional property to be set in the servers *standalone.xml* file (JBoss Inc., 2015b):

```
1 <subsystem xmlns="urn:jboss:domain:logging:2.0">
2    <add-logging-api-dependencies value="false"/>
3       .
4       .
5 </subsystem>
```

Listing 4.17: Turing off WildFly's inherent log4j module (standalone.xml)

In this manner, WildFly is instructed to let deployments configure logging on their own and the settings defined in *SherLog's log4j.properties* file are applied as expected.

4.5 Connecting to *SherLog* with JConsole

Connecting to *SherLog* works in exactly the same way as the document demonstrated for the *Watson*. After having launched the server process along with the *SherLog* agent, JConsole can be used to analyze the debugging tool's MBean. Figure 4.3 presents a screenshot that illustrates how the examination of the MBeans characteristics might look like:

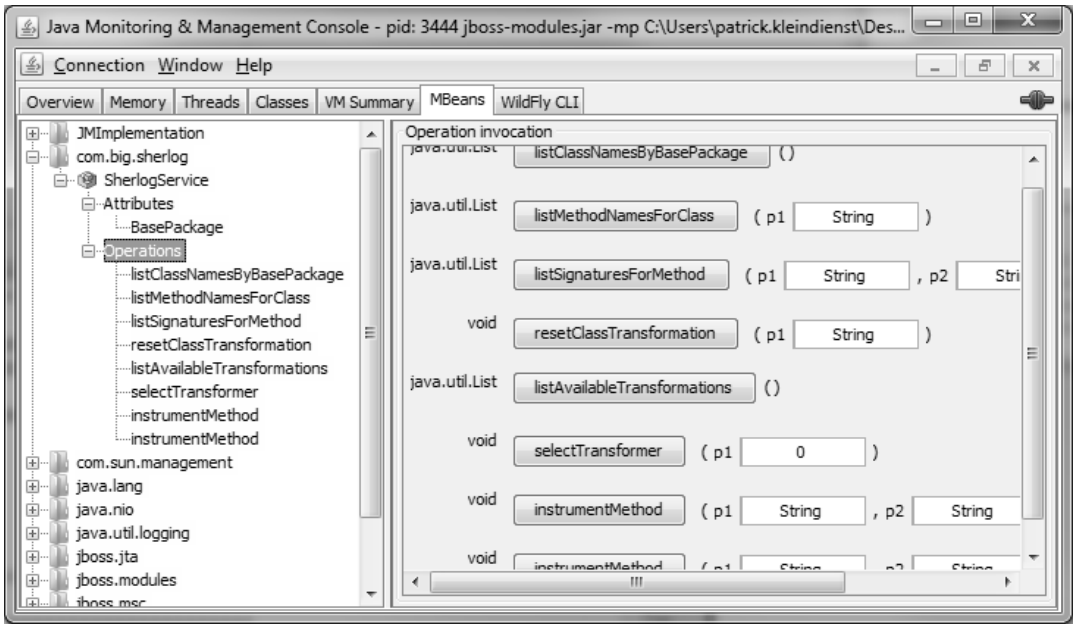

Figure 4.3: Examining the *SherLog* MBean with JConsole

The *SherLog* debugging tool is now ready for instrumenting application classes. Its output can be found under the current user's home directory, where it will create a *sherlog* directory containing a corresponding log file.

Chapter 5

Evaluation

After having demonstrated two possible solutions to implement the specification established in chapter 1, the document's next part focuses on the assessment of both *Watson* and *SherLog*. The goal is to summarize the characteristics of the tools and to analyze which of the demands are met.

5.1 *Watson*: Validation of requirements

5.1.1 Fulfilled conditions

At first, the evaluation steps through the criteria the *Watson* debugging tool meets to the specification's satisfaction:

- *Installation*: As shown in section 3.5, setting up *Watson* requires to put its JAR file on an application's class path and to add the configuration bean to the Spring context. Henceforward, the tool integrates seamlessly into existing environments. Indeed, the preparation takes a already running webapp to be stopped and restarted afterwards. However, this can be considered a reasonable cost in combination with the simple configuration, since it has to be done just once.

- *Granularity*: Although Spring AOP offers a limited extent of available interception points (Walls, 2014, pp. 102-103), the framework's support of stepping into method executions perfectly corresponds to the specification.

- *Remoteness*: *Watson* relies on the JMX technology for facilitating the communication with the tool residing on the server. Due to the provided abstraction layer on top of the actual communication protocol, a user does not notice a difference between working with the tool on a local or a remote host (Oracle Corporation, 2015j).
  Establishing a connection merely requires knowledge about the server's *Internet Protocol (IP)* address as well as the respective port number, assumed that a company's network infrastructure does not specify any restrictions concerning a machine's accessibility.

- *Security*: For preventing an application from being contaminated with malicious code, *Watson* does not allow to type instrumentation code directly into an input mask. Instead, it only approves the employment of predefined components, either provided by the user or the tool itself.
  Moreover, since JMX offers built-in support for authentication as well as the encryption of network communication (Oracle Corporation, 2015l), the thesis is comfortable with *Watson's* degree of security.

- *Extensibility*: Another important criteria demands the tool to allow user-defined customizations. *Watson* offers this kind of flexibility by providing extension points in the form of dedicated interfaces. Furthermore, a auto-discovery mechanism keeps a user from encounter the debugging tool's code base.

5.1.2 Partially fulfilled or missed conditions

In addition to *Watson's* satisfying characteristics, it also features different deficiencies:

- *Installation*: The setup must also be considered when looking at present handicaps. Although the basic configuration is simple, instrumenting a certain class requires it to be annotated with *Watson's* marker annotation (see section 3.3.5). Indeed, generously annotating application classes at development time constitutes an arguable effort. However, having to stop a webapp as well as the server because of classes that have been ignored before the deployment may be annoying.

- *Platforms*: *Watson's* dependency on Spring's AOP and JMX projects results in the implication that it cannot be installed in environments which are not based on Spring. One the one hand, it benefits from the features that already ship with the Spring ecosystem. But on the other hand, this limits its field of application.

- *Handling*: Although *Watson's* operational interface offers an intuitive handling by enabling a user to instrument classes without knowledge about an application's project structure, operating the tool on behalf of the JConsole GUI is not very comfortable. In contrast to the enterprise platform contraints, there definitely are possibilities to sort out this kind of problem. Chapter 6 will demonstrate how a possible solution might look like.

5.2 *SherLog*: Validation of requirements

5.2.1 Fulfilled conditions

Like the evaluation of the first implementation, the assessment of the *SherLog* debugging tool starts with the properties that correspond to the specification:

- *Installation*: The cost for setting up *SherLog* is comparable to *Watson's*. Foremost, its JAR file has to be added to a webapp's classpath. Then, it has to be launched as a Java agent as soon as the underlying application server is started (Oracle Corporation, 2015h). At this point, it is accessible via JMX.

- *Granularity*: *SherLog* also answers the requirement on method interception without any difficulties. As this implementation rests upon Javassist for instrumentation, it can profit from the library's capabilities for achieving fine-grained bytecode enrichment (JBoss Inc., 2015a).

- *Remoteness*: Since *SherLog* and *Watson* make use of the same communication technology, there is no difference between these tools as far as their local or remote control is concerned.

- *Security*: Manipulating code on the bytecode level introduces the risk of code injection attacks. Therefore, *SherLog* also defines high-level APIs for the instrumentation instead of offering any interfaces for directly transfering source code to an application.
  In addition, the Instrumentation API protects the application's class definitions from being altered in a critical way (Oracle Corporation, 2015e).
  As for authentication and the encryption of the communication, *SherLog* delegates the responsibility to JMX as well (Oracle Corporation, 2015l).

- *Extensibility*: When it comes to customization, *SherLog* is very similar to *Watson* again. It uses the same interface-based extension principle and automatically integrates user-defined implementations.

- *Platforms*: Unlike *Watson*, the *SherLog* debugging tool is platform-agnostic. This means that it does not depend on any frameworks or libraries which are specific to a certain Java enterprise environment. As a consequence of this autonomy, *SherLog* can be installed in application servers conforming with the Java EE standard as well as in servlet containers supplying Spring-based webapps.

5.2.2 Partially fulfilled conditions

The existence of several drawbacks also holds good for *SherLog*:

- *Installation*: Having to examine the setup process from different perspectives also applies to *SherLog*. Although the fundamental configuration is not very complex, section 4.4 demonstrated the different changes that were necessary to make *SherLog* work properly in the context of a WildFly server. Though, as these adoptions are demanded by the server infrastructure, their extent may vary across the different application server vendors.

- *Handling*: So far, interacting with *SherLog* is likewise restricted to the usage of JConsole. The resort shown in the next chapter constitutes an approach for improving the usability of both tools.

5.3 Conclusion

The assessment of the outlined implementations reveals that neither *Watson* nor *SherLog* can comply with the initial specification entirely. Nevertheless, the aspects that differ from the thesis' expectations do not prevent both tools from representing a practicable supplement in a developer's workaday life. So, the main purpose of this thesis can be considered fulfilled.
The issue whether to use *SherLog* or *Watson* depends on a project's general conditions as well as the needs of a single developer or the whole team. All in all, *Watson* poses a good starting point when developing a Spring application. In that case, this tool's platform restrictions are not relevant and the user saves the effort to deal with Java agents and possible sever configuration issues.
In contrast, using *SherLog* makes sense if an application must not depend on any Java enterprise standards. It possibly requires the adaption of the application server's configuration, but also offers significant flexibility afterwards.
Furthermore, *SherLog* can profit by the Instrumentation API as an additional barrier against the infiltration of malicious code, whereas *Watson's* level of security rather depends on reasonable extension implementations. But otherwise, *Watson* has the advantage of being more like a high-level approach, avoiding to interact with the JVM's internals and therefore decreasing the risk of serious damage.

Chapter 6

Outlook

The last part of the thesis glances at further improvements concerning the current state of the introduced debugging tools.

6.1 Building a custom JMX client with JavaFX

Although the server-side implementations of *Watson* and *SherLog* conform to the requirements on a grand scale, the evaluation already underlined their lack of usablity and convenience. Operating them with the JConsole GUI results in lots of copying and pasting of Strings representing the names of classes, variables or methods. Besides, the appearence of JConsole cannot really be judged as clear and pleasant. Thus, the project plan provides the development of a custom JMX client for making the usage of both debugging tools more comfortable.
For this purpose, *JavaFX* poses a suitable starting point. This is a technology which allows the implementation of rich client applications based on plain Java as well as *Cascading Style Sheets (CSS)* (Oracle Corporation, 2015a).
Since JavaFX is designed as a Java API, it can make use of any other Java library like the JMX Remote API. This library contains classes which enable a developer to build his own JMX client applications and connect them to local or remote MBean servers (Oracle Corporation, 2015j).

6.2 Further improvements

Besides the development of a custom JMX client, there are also several minor enhancements that are supposed to be realized in the future.
One of these issues is to upgrade the log4j version used by both debugging tools to log4j 2.x (Apache Software Foundation, 2015e). Although they work perfectly with log4j 1.2 this is a reasonable adaption, since Apache announced the end of life for log4j 1, as Thomas (2015) explains in his *JAXenter* article. He points out that all users are recommended to upgrade their projects as soon as possible (Thomas, 2015).
In addition, both *Watson* and *SherLog* are inteded to be equipped with further instrumentation logic that can be used out of the box.

List of Figures

Listings

Bibliography

AOP Alliance (2015a). *MethodInterceptor*. URL: http://aopalliance.sourceforge.net/doc/org/aopalliance/intercept/MethodInterceptor.html (visited on 07/27/2015).

— (2015b). *MethodInvocation*. URL: http://aopalliance.sourceforge.net/doc/org/aopalliance/intercept/MethodInvocation.html (visited on 08/05/2015).

Apache Software Foundation (2015a). *Apache log4j-1.2*. URL: http://logging.apache.org/log4j/1.2/ (visited on 07/01/2015).

— (2015b). *Apache Maven Assembly Plugin Introduction*. URL: http://maven.apache.org/plugins/maven-assembly-plugin/ (visited on 07/09/2015).

— (2015c). *Apache Tomcat - Welcome!* URL: http://tomcat.apache.org/ (visited on 07/04/2015).

— (2015d). *Apache Tomcat 7 (7.0.63) - Tomcat Web Application Deployment*. URL: https://tomcat.apache.org/tomcat-7.0-doc/deployer-howto.html (visited on 07/29/2015).

— (2015e). *Log4j Log4j 2 Guide - Apache Log4j 2*. URL: http://logging.apache.org/log4j/2.x/ (visited on 08/17/2015).

— (2015f). *Maven Welcome to Apache Maven*. URL: https://maven.apache.org/ (visited on 07/01/2015).

Bailey, John, David M. Lloyd, and Thomas Diesler (2014). *jboss-modules/jboss-modules*. GitHub. URL: https://github.com/jboss-modules/jboss-modules (visited on 08/17/2015).

Beams, Chris (2015). *Aware (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/Aware.html (visited on 07/21/2015).

Bruneton, Eric (2014). *ClassVisitor (ASM 5.0 Documentation)*. URL: http://asm.ow2.org/asm50/javadoc/user/org/objectweb/asm/ClassVisitor.html (visited on 08/05/2015).

Chiba, Shigeru (2015). *Javassist*. URL: http://jboss-javassist.github.io/javassist/ (visited on 07/01/2015).

Eclipse Foundation (2003). *Chapter 5. Load-Time Weaving*. URL: https://eclipse.org/aspectj/doc/released/devguide/ltw.html (visited on 08/04/2015).

— (2015). *The AspectJ Project*. URL: https://eclipse.org/aspectj/ (visited on 07/01/2015).

Gamma, Erich, Richard Helm, and Ralph Johnson (2001). *Entwurfsmuster . Elemente wiederverwendbarer objektorientierter Software*. 2. Aufl. Muenchen: Addison-Wesley. 484 pp.

Gierke, Oliver (2015). *Von einem SSpring versus JavaEEßu reden, ist faktisch eigentlich falsch*. jaxenter.de. URL: https://jaxenter.de/von-einem-spring-versus-javaee-zu-reden-ist-faktisch-eigentlich-falsch-20965 (visited on 07/04/2015).

Guelcue, Ceki (2012). *FileAppender (Apache Log4j 1.2.17 API)*. URL: https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/FileAppender.html (visited on 08/15/2015).

Haase, Arno (2015). *Classpath-Scan im Eigenbau [Aus der Java-Trickkiste]*. jaxenter.de. URL: https://jaxenter.de/classpath-scan-im-eigenbau-aus-der-java-trickkiste-12963 (visited on 07/13/2015).

Hoeller, Juergen (2015a). *BeanFactoryAspectJAdvisorsBuilder (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/4.1.7.RELEASE/javadoc-api/ (visited on 08/08/2015).

— (2015b). *BeanPostProcessor (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/4.1.7.RELEASE/javadoc-api/ (visited on 08/08/2015).

Hoeller, Juergen and Rob Harrop (2015a). *BeanDefinition (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanDefinition.html (visited on 07/19/2015).

— (2015b). *NameMatchMethodPointcutAdvisor (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/4.1.7.RELEASE/javadoc-api/ (visited on 08/08/2015).

Hoeller, Juergen, Rod Johnson, and Rob Harrop (2015). *AbstractAutoProxyCreator (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/aop/framework/autoproxy/AbstractAutoProxyCreator.html (visited on 07/24/2015).

JBoss Inc. (2015a). *Javassist Tutorial*. URL: http://jboss-javassist.github.io/javassist/tutorial/tutorial.html (visited on 07/05/2015).

— (2015b). *Logging Configuration - WildFly 8 - Project Documentation Editor*. URL: https://docs.jboss.org/author/display/WFLY8/Logging+Configuration (visited on 07/17/2015).

JetBrains s.r.o. (2015). *IntelliJ IDEA 14.1.0 Help :: Maven*. URL: https://www.jetbrains.com/idea/help/maven.html (visited on 07/29/2015).

Johnson, Rod (2004). *Advice*. URL: http://aopalliance.sourceforge.net/doc/org/aopalliance/aop/Advice.html (visited on 08/05/2015).

Johnson, Rod and Juergen Hoeller (2015a). *Advised (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/aop/framework/Advised.html (visited on 07/21/2015).

— (2015b). *AdvisedSupport (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/aop/framework/AdvisedSupport.html (visited on 07/21/2015).

— (2015c). *AnnotationAwareAspectJAutoProxyCreator (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/aop/aspectj/annotation/AnnotationAwareAspectJAutoProxyCreator.html (visited on 07/27/2015).

— (2015d). *AopProxyUtils (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/aop/framework/AopProxyUtils.html (visited on 07/21/2015).

— (2015e). *JdkDynamicAopProxy (Spring Framework)*. URL: http://docs.spring.io/spring/docs/1.2.x/api/org/springframework/aop/framework/JdkDynamicAopProxy.html (visited on 07/24/2015).

Johnson, Rod, Juergen Hoeller, and Chris Beams (2015). *ApplicationContextAware (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/ApplicationContextAware.html (visited on 07/21/2015).

Johnson, Rod, Juergen Hoeller, and Adrian Colyer (2015). *ReflectiveMethodInvocation (Spring Framework 4.1.7.RELEASE API)*. URL: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/aop/framework/ReflectiveMethodInvocation.html (visited on 07/26/2015).

Johnson, Rod et al. (2015). *CglibAopProxy*. URL: http://docs.spring.io/spring/docs/4.0.0.M1_to_4.2.0.M2/Spring%20Framework%

204.0.0.M1/org/springframework/aop/framework/CglibAopProxy.html (visited on 07/26/2015).

Lahres, Bernhard and Gregor Raman (2015). *Rheinwerk Computing :: Praxisbuch Objektorientierung 9.2 Aspektorientiertes Programmieren*. URL: http://openbook.rheinwerk-verlag.de/oo/oo_07_aspekteinoo_001.htm#Rxxob07aspekteinoo001040016261f012130 (visited on 07/02/2015).

Liang, Sheng and Gilad Bracha (1998). "Dynamic Class Loading in the Java Virtual Machine". In: p. 9. (Visited on 07/08/2015).

Microsoft Corporation (2015). *Herstellen einer Verbindung mit einem anderen Computer mithilfe der Remotedesktopverbindung - Windows-Hilfe*. windows.microsoft.com. URL: http://windows.microsoft.com/de-de/windows/connect-using-remote-desktop-connection (visited on 08/06/2015).

Oracle Corporation (2004). *GC: InstrumentationImpl - sun.instrument.InstrumentationImpl (.java) - GrepCode Class Source*. URL: http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/sun/instrument/InstrumentationImpl.java (visited on 07/28/2015).

— (2014a). *JAR File Specification*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html (visited on 08/17/2015).

— (2014b). *jdb - The Java Debugger*. URL: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html (visited on 07/29/2015).

— (2015a). *1 JavaFX Overview (Release 8)*. URL: http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784 (visited on 08/10/2015).

— (2015b). *ClassFileTransformer (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/ClassFileTransformer.html (visited on 07/06/2015).

— (2015c). *HashMap (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html (visited on 08/05/2015).

— (2015d). *InputStream (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html (visited on 08/05/2015).

— (2015e). *Instrumentation (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html (visited on 07/05/2015).

Oracle Corporation (2015f). *Java EE - Technologies | Oracle Technology Network | Oracle*. URL: http://www.oracle.com/technetwork/java/javaee/tech/index.html (visited on 06/29/2015).

— (2015g). *Java Platform, Enterprise Edition (Java EE) | Oracle Technology Network | Oracle*. URL: http://www.oracle.com/technetwork/java/javaee/overview/index.html (visited on 06/29/2015).

— (2015h). *java.lang.instrument (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html (visited on 07/05/2015).

— (2015i). *Lesson: Members (The Java Tutorials > The Reflection API)*. URL: https://docs.oracle.com/javase/tutorial/reflect/member/index.html (visited on 07/22/2015).

— (2015j). *Lesson: Overview of the JMX Technology (The Java Tutorials > Java Management Extensions (JMX))*. URL: https://docs.oracle.com/javase/tutorial/jmx/overview/ (visited on 07/01/2015).

— (2015k). *Map (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Map.html (visited on 08/05/2015).

— (2015l). *Monitoring and Management Using JMX Technology - Java SE Monitoring and Management Guide*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/management/agent.html (visited on 08/07/2015).

— (2015m). *Object (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html (visited on 07/23/2015).

— (2015n). *Packaging Applications - The Java EE 6 Tutorial*. URL: http://docs.oracle.com/javaee/6/tutorial/doc/bnaby.html (visited on 07/29/2015).

— (2015o). *Standard MBeans (The Java Tutorials > Java Management Extensions (JMX) > Introducing MBeans)*. URL: https://docs.oracle.com/javase/tutorial/jmx/mbeans/standard.html (visited on 07/06/2015).

— (2015p). *The Java Virtual Machine Specification*. URL: http://docs.oracle.com/javase/specs/jvms/se8/html/ (visited on 07/04/2015).

— (2015q). *Trail: RMI (The Java Tutorials)*. URL: https://docs.oracle.com/javase/tutorial/rmi/ (visited on 07/28/2015).

— (2015r). *VisualVM*. URL: http://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/ (visited on 08/09/2015).

OW2 Consortium (2015). *ASM - Home Page*. URL: http://asm.ow2.org/ (visited on 07/10/2015).

Pivotal Software Inc. (2015a). *10.Spring AOP APIs*. URL: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop-api.html (visited on 07/22/2015).

— (2015b). *5.The IoC container*. URL: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-factory-scopes (visited on 07/19/2015).

— (2015c). *5.The IoC container*. URL: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-factory-extension-factory-postprocessors (visited on 07/19/2015).

— (2015d). *9.Aspect Oriented Programming with Spring*. URL: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html (visited on 07/02/2015).

— (2015e). *Projects*. URL: https://spring.io/projects (visited on 06/29/2015).

— (2015f). *Spring Boot*. URL: http://projects.spring.io/spring-boot/ (visited on 07/24/2015).

— (2015g). *spring.io*. URL: https://spring.io/ (visited on 08/06/2015).

Red Hat Inc. (2015a). *Byteman - JBoss Community*. URL: http://byteman.jboss.org/ (visited on 08/02/2015).

— (2015b). *Class Loading in WildFly - WildFly 8 - Project Documentation Editor*. URL: https://docs.jboss.org/author/display/WFLY8/Class+Loading+in+WildFly?_sscc=t (visited on 07/07/2015).

— (2015c). *Red Hat Customer Portal*. URL: https://access.redhat.com/documentation/en-US/JBoss_Enterprise_Application_Platform/6/html/Development_Guide/chap-Class_Loading_and_Modules.html (visited on 07/13/2015).

Thomas, Michael (2015). *Apache gibt End of Life von log4j 1 bekannt*. jaxenter.de. URL: https://jaxenter.de/apache-gibt-end-of-life-von-log4j-1-bekannt-24533 (visited on 08/10/2015).

Ullenboom, Christian (2011). *Rheinwerk Computing :: Java ist auch eine Insel - 1 Java ist auch eine Sprache*. URL: http://openbook.rheinwerk-verlag.de/javainsel/javainsel_01_002.html#dodtpaef1ef41-da01-4368-8372-4f815650cf09 (visited on 07/29/2015).

Walls, Craig (2014). *Spring in Action*. 4th ed. Manning Pubn.

ZeroTurnaround (2015). *JRebel Java Plugin: Eclipse, IntelliJ, NetBeans | zeroturnaround.com*. ZeroTurnaround. URL: http://zeroturnaround.com/software/jrebel/ (visited on 08/02/2015).